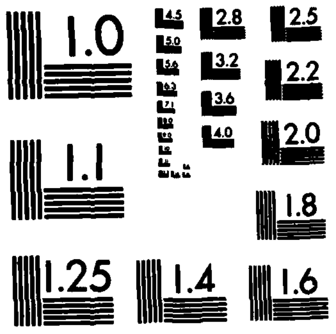
AD-A123 305 LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MES..(U) GENERAL DYNAMICS FORT WORTH TX DATA SYSTEMS DIV 09 NOV 82 DAAK80-81-C-0108 1/2

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0

1

AD A123305

CASE STUDY I

FINAL REPORT DEVELOPED FOR LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MESSAGE SWITCH USING THE ADA PROGRAMMING LANGUAGE

U. S. ARMY CECOM
CONTRACT NO. DAAK80-81-C-0108

VOLUME II OF IV

GENERAL DYNAMICS
DATA SYSTEMS DIVISION
CENTRAL CENTER
P. O. BOX 748
FORT WORTH, TX 76101

83 01 12 033

50272-101

REPORT DOCUMENTATION PAGE	1. REPORT NO.	2.	3. Recipient's Accession No. AD-A123 305
4. Title and Subtitle Ada Capability Study: Design of the Message Switching System AN/TYC-39 Using the Ada Programming Language			5. Report Date 9 November 1982 6.
7. Author(s) General Dynamics			8. Performing Organization Rept. No.
9. Performing Organization Name and Address General Dynamics Data Systems Division Central Center P. O. Box 748 Fort Worth, TX 76101			10. Project/Task/Work Unit No. 11. Contract(C) or Grant(G) No. (C) DAAK80-81-C-0108 (G)
12. Sponsoring Organization Name and Address USA CECOM Center for Tactical Computer Systems (CENTACS) ATTN: DRSEL-TCS-ADA-1 Fort Monmouth, NJ 07703			13. Type of Report & Period Covered Final 14.

15. Supplementary Notes

16. Abstract (Limit: 200 words)

An Ada oriented framework for the design and documentation of the U. S. Army TYC-39 store and forward message switch (military software) system is presented. This document package contains a Requirements, Design, Ada Integrated Methodology, and Final Report section. A methodology to use Ada in specifying requirements, design, and the implementation of a system was developed. This methodology was used to redesign the TYC-39 message switch system. A selected software module was programmed after the redesign.

17. Document Analysis a. Descriptors

Ada Programming Language
Software Design with Ada
Designing with Ada

b. Identifiers/Open-Ended Terms

Message Switch
Military Software
Program Design Language

c. COSATI Field/Group

18. Availability Statement: ~~Distribution limited to the United States~~. Available from National Technical Information Service, Springfield, VA 22161.	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 521
	20. Security Class (This Page) UNCLASSIFIED	22. Price

CASE STUDY I

FINAL REPORT DEVELOPED FOR LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MESSAGE SWITCH USING THE ADA PROGRAMMING LANGUAGE

U. S. ARMY CECOM
CONTRACT NO. DAAK80-81-C-0108

VOLUME II OF IV

GENERAL DYNAMICS
DATA SYSTEMS DIVISION
CENTRAL CENTER
P. O. BOX 748
FORT WORTH, TX 76101

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

The Ada Integrated Methodology (AIM) was developed as a product of the Ada Capability Study that was conducted at General Dynamics Data Systems Division. This study was performed in response to a contract that was awarded to General Dynamics by the United States Department of Defense (DoD). The purpose of the Ada Capability Study was to provide data/information to the DoD that would be useful in formulating an Ada-based communications system methodology and an Ada training program.

1.2 NATURE OF THE ADA CAPABILITY STUDY

The structure of the study is illustrated in Figure 1.2-1. Within this structure, the methodology team developed AIM ich was applied to the redesign of a message switch system by design team. The redesign was a real-world case study bec e the system redesigned is currently in production.

Problems associated with the application of AIM to the red gn of the message switch system were documented by the design team. Resolutions of these problems resulted in changes to AIM throughout the project. AIM, along with the problems documented and their resolutions were delivered to the DoD at the conclusion of the study. The intent was to make this information available to the DoD as input for making decisions relating to the use of Ada in the future.

1.3 OVERVIEW OF AIM

AIM is a requirements and design methodology developed specifically for application to the development of Ada-based communications systems. It integrates several existing methodologies and some important design concepts with the power of the Ada language.

1.3.1 Scope

AIM was being tailored for (1) communication systems and (2) Ada constructs. Characteristics of a message switch system were studied in an effort to facilitate AIM sensitivity to communication systems. Additionally, Ada constructs were studied extensively for incorporation into AIM. However, not only Ada constructs themselves, but concepts such as information hiding and abstract data types which are supported by Ada were also studied.

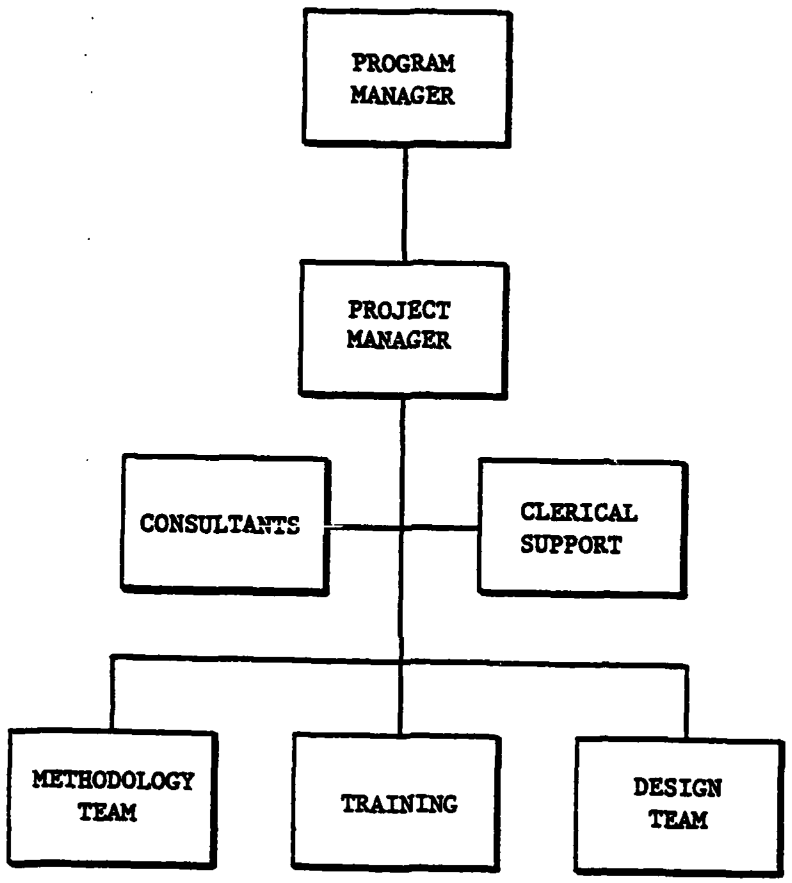


Figure 1.2-1 Ada Capability Study Structure.

There are many elements of existing methodologies which are beyond the scope of this methodology development effort. Specifically, the methodology excludes the following:

- Development of documentation and plans in support of the implementation of a new system such as (1) user guide, (2) operations guide, (3) test plan, (4) new procedures, and (5) training aids and courses
- Survey of present system
- Conversion from present system
- Implementation phase (does not address programming and testing) - goes through design only
- Constraints such as time, money, and personnel
- Detailed explanations of contributing methodologies, concepts, and the Ada language (provides only basic guidelines and examples for the requirements analyst and designer).

The AIM project life cycle concept can be used to help illustrate the scope of AIM (see Figure 1.3.1-1). The dotted lines on Figure 1.3.1-1 indicate where AIM begins and ends. Although there is no implementation methodology within AIM, Chapter 4 does provide some Ada development standards.

1.3.2 Description of Ada Requirements Methodology (ARM)

ARM is a requirements methodology for mapping A-specifications (for communications systems) into requirments that will be used as input to the design process (see Figure 1.3.1-1). The application of ARM results in a set of system requirements expressed in three basic formats - (1) graphic illustrations (models of the system functions, logical data structures, and concurrency), (2) structured English which uses the Ada language constructs as a base that is augmented by a disciplined use of the English language, and (3) English narrative.

The purpose of ARM is to effect an understanding of the problem. The requirements analyst must understand the problem and convey this understanding to the designer via the Ada Requirements Document produced by the application of ARM.

Three basic assumptions were made prior to the development of ARM. First, it was assumed that the A-specification document needed to be decomposed and rewritten in a more structured format with graphic illustrations in order to eliminate ambiguity and enhance clarity. Secondly, it was assumed that the requirements analyst and designer are not the same person. Thirdly, the Ada language was assumed to be the implementation language.

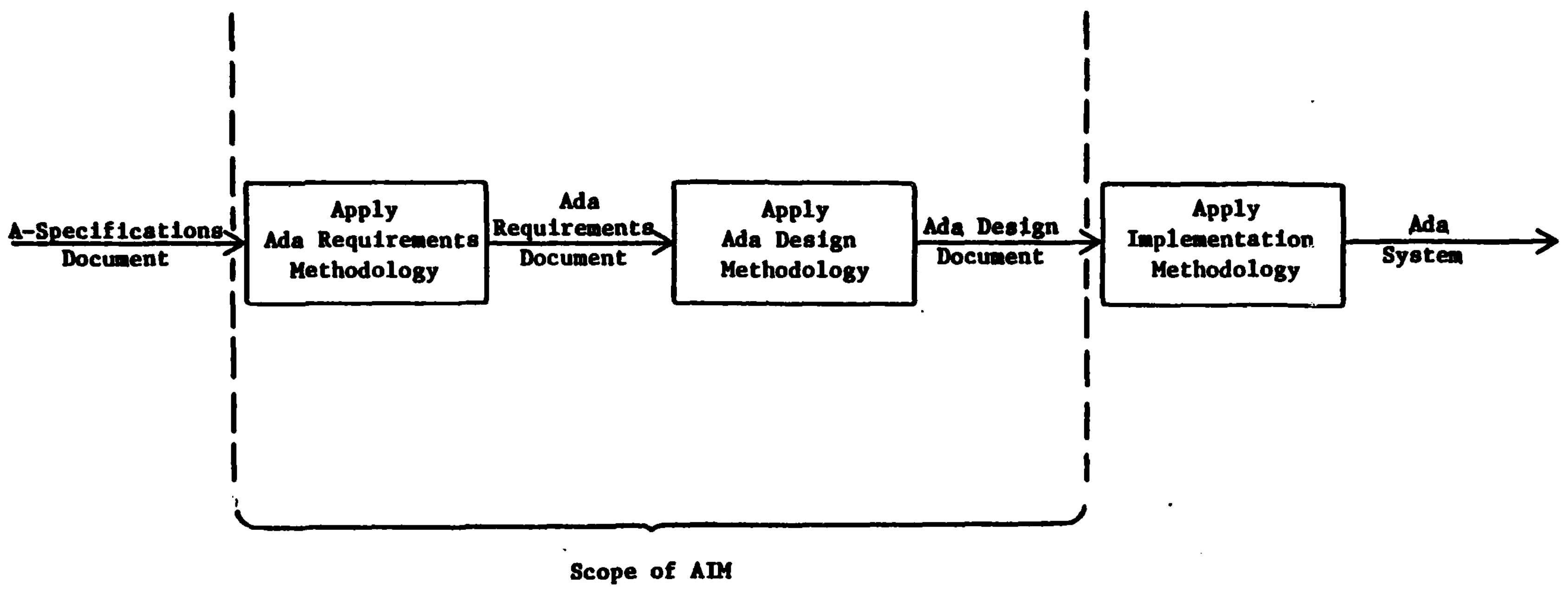

A-Specifications Document
Apply Ada Requirements Methodology
Ada Requirements Document
Apply Ada Design Methodology
Ada Design Document
Apply Implementation Methodology
Ada System
Scope of AIM

Figure 1.3.1-1 AIM Project Life Cycle Concept.

1.3.2.1 Derivation

ARM was developed mainly from two existing methodologies - Structured Analysis (Yourdon and De Marco) and SADT (Softech). Both of these methodologies require functional decomposition of the problem being studied during the analysis or requirements phase of system development. Functional decomposition might even be considered as the nucleus of both methodologies. Therefore, ARM is heavily oriented toward functional decompcsition. Also the paper, "Software Requirements: A Report on the State of the Art", pp. 22-30; Yeh, Zave, Conn, and Cole; October 1980, was the primary contributor in the non-functional requirements area of ARM.

1.3.2.1.1 Contribution of Structured Analysis

Structured analysis is a formal methodology for developing system specifications or requirements that feed into the design process. As stated above, functional decomposition is the salient feature of Structured Analysis. A data flow diagram (DFD) is the vehicle used to facilitate functional decomposition. Specifically, a high-level system diagram that shows all the major functions of the system is decomposed (by function) until all major functions have been broken down sufficiently for understanding. This method of functional decomposition is employed by ARM.

The DFDs created by the application of structured analysis show functions and their interfaces (data flows and files) with each other. ARM uses DFDs in this same manner with some modifications from SADT (which are addressed in a subsequent section below).

Structured analysis also requires the development of a data dictionary, logical data structure models, and structured English specifications that describe the functions illustrated by the lowest-level DFDs. ARM requires the development of these same components. However, instead of structured English, ARM uses a combination of Ada language constructs and a disciplined version of the English language to state function specifications or requirements.

1.3.2.1.2 Contribution of SADT

SADT employs some mechanisms during the development of DFDs that have been adopted by ARM. Rectangular blocks are used to symbolize functions on SADT diagrams instead of the circular mechanism used by Structured Analysis to illustrate functions. ARM employs SADT's rectangular block representation of functions during the development of DFDs.

SADT uses the four sides of the block structures (functions) to distinguish between inputs, outputs, control variables, and man/machine interfaces. ARM employs this concept also.

Concurrency is not included in SADT DFDs. However, SADT does recommend addressing concurrency after developing the DFDs for the system. ARM incorporates this same philosophy.

1.3.2.1.3 Contribution of Ada

Even though functional decomposition is the salient feature of ARM and the main vehicle for producing an understanding of the problem, the use of Ada as a requirements language is the first step toward using Ada throughout the system development life cycle. The use of Ada as a program design language (PDL) during design will capitalize on the capabilities of Ada even more than its use as a requirements language.

Ada has the constructs that support writing concise, unambiguous, structured specifications. Therefore, a subset of the Ada language (which includes the necessary constructs) formed the basis for a structured requirements specification language to be used for writing functional specifications. The subset of the Ada constructs utilized by ARM is provided below:

if statement
case statement
loop statement
assignment statement
code statement
expression
relation
exit statement
procedure calls for pre-defined procedures
raise statement
exception handler.

As the requirements specifications language, Ada is used to express the specifications for each lowest level (primitive) function of the DFD. Disciplined English is used to supplement the Ada requirements language constructs when they are insufficient for representing a particular requirement. Additionally, comments are used where appropriate to add clarity to the requirements. The symbol for a comment line is "--", the same as the Ada comment notation. A "-->" symbol is used to indicate a line of requirements specifications that is stated in disciplined English. Figure 1.3.2-1 is an example of an Ada requirements specification which uses Ada constructs, disciplined English, and comments.

The use of Ada as the tool for expressing functional requirements is expected to provide a high degree of continuity from requirements to design. This, of course, should result in fewer conversions from requirements to design specifications.

```
DFD DIAGRAM:  A344  MODIFY MCB
A-SPECIFICATIONS REFERENCE:  PARAGRAPHS 3.2.1.2.13.1, DCAC-370-D175-1
                             TABLE 8-1
DESCRIPTION:  EXAMINE A RECEIVED MESSAGE CONTROL BLOCK (MCB) TO DETERMINE IF THE
              MESSAGE IS ORBITING; IF NOT, THEN MODIFY AS REQUIRED TO PREVENT
              ORBITING, ADD THE TIME OF MESSAGE RECEIPT, AND RECALCULATE BLOCK
              PARITY (BP).  IF MESSAGE IS ORBITING, NOTIFY OPERATOR.
REVISION:  BAAA
DATE:  12/14/81
AUTHOR:  P. DOBBS

SPECIFICATIONS:
  type CT is 0..2;
  COUNT : CT;
begin
  COUNT :=0;
  for all CHARACTERS in TDW loop
    if CHARACTER = SWITCH ID then
      COUNT := COUNT + 1;
    end if;
  end loop;
  case COUNT is
    when 0 =>
      --> ADD SWITCH_ID TO FIRST ZERO FIELD
    when 1 =>
      -->  ZERO ALL FIELDS AFTER POSITION OF SWITCH_ID
      -->  ADD SWITCH_ID SECOND TIME
    when 2 =>
      -->  HOLD MESSAGE
      --   PROVIDE ORBIT INFORMATION
      -->  NOTIFY OPERATOR
  end case;
    --> ADD SOH_TIME
  BP := MCB(1);
  for I in 2..83 loop
    BP := BP xor MCB(I);
  end loop;
end MODIFY_MCB;
```

Figure 1.3.2-1 Example of Ada Requirements Specification

1.3.2.2 Scope of ARM

ARM restates and graphically illustrates system requirements in the A-specifications document (both functional and non-functional). It is not intended to constrain or limit the designer. However, the functional decomposition and creation of DFDs in the requirements phase is considered by some as moving toward design. The intent of ARM is to produce an understanding of the problem and a requirements document that will aid and facilitate the design process. The inclusion of functional decomposition and DFDs in ARM is consistent with a trend in contemporary system development to incorporate more planning, discipline, and decision-making up front (prior to program development).

1.3.2.3 Outputs

The output of ARM is an Ada Requirements Document. Components of this document are listed below:

a. Functional Model of System (Data Flow Diagram)

b. Ada Function Requirements

c. Data Dictionary

d. Logical Data Structures Model

e. Concurrency Model(s)

f. Non-Functional Requirements

The Ada Requirements Document is the vehicle for conveying system requirements to the designer and is intended to feed into and influence the design process.

1.3.3 Description of Ada Design Methodology (ADM)

ADM is a design methodology that converts the output of ARM (Ada Requirements Document) to a system design (see Figure 1.3.1-1). The application of ADM results in a system design expressed by graphic models and Ada program design language (PDL).

Several methodologies and design concepts have been combined with the Ada language to form ADM. Therefore, the designer that applies ADM should be equipped with a design "tool bag".

The purpose of ADM is two-fold. First, ADM must produce a design that satisfies the requirements in the Ada Requirements Document. Secondly, it must produce a design that exhibits maintainability (flexibility for design changes during development and after implementation). Four assumptions influenced the development of ADM. These assumptions are listed below:

a. The application of ARM produces an Ada Requirements Document that supports and influences a structured design process.

b. The implementation language is Ada.

c. The design must, above all, satisfy the system requirements as stated in the Ada Requirements Document.

d. Maintainability is the theme that must permeate the design process.

1.3.3.1 Derivation

As stated above, ADM is a combination of selected design methodologies, concepts, and the Ada language. As a result, there were many contributors to its development.

1.3.3.1.1 Contribution of Object-Oriented Design (Booch and Jackson)

The Object-Oriented Design Methodology as suggested by Grady Booch in his article, "Describing Software Design in Ada", has been employed by ADM as the first step of the design process. Some concepts from Michael Jackson's data-driven design approach have been merged with Booch's object-oriented design approach in this step.

The intent of the object-oriented design step is to take a first cut at identifying the high-level objects of the system and establishing Ada packages around these objects that exhibit informational strength. However, object-oriented design is mostly an art and has not been formalized as a mature design methodology. As a result, the packages established during the application of object-oriented design are not immediately incorporated into the system design. Instead, they are put on hold and reconciled with the system design produced later in the design phase.

1.3.3.1.2 Contribution of Structured Design (Yourdon and Constantine)

Structured Design contributed heavily to the development of ADM. A structure chart (an important structured design tool) is produced from the DFD produced during the requirements phase using Structured Design techniques. Once the initial structure chart is produced, it is evaluated in terms of goodness using two important structured design concepts (coupling and cohesion). Additionally, some Structured Design heuristics are used to produce an initial structure chart and then to evaluate and enhance it until a satisfactory functional design is achieved.

1.3.3.1.3 Contribution of Composite Design (Myers)

The main contribution of Composite Design to the development of ADM was to augment the Structured Design approach utilized. Myers' Composite Design approach embraces many of the same principles of Structured Design with a little different flavor. Perhaps the most significant contribution of Composite Design is a cohesion concept called information strength which supports information hiding and maintainability.

1.3.3.1.4 Contribution of Data-Driven Design Approach (Jackson)

Michael Jackson's book, Principles of Program Design, describes the Data-Driven Design approach which is used at two points during the application of ADM. First, Jackson's approach is used to facilitate object-oriented design. System objects are represented using Jackson's mechanism for representing data structures, and operations are assigned to objects. Secondly, the Data-Driven Design approach may be applied to subproblems during the development of a structure chart.

1.3.3.1.5 Contribution of Selected Design Concepts

There are two concepts that are very basic to design - coupling and cohesion. Coupling is a measure of the quality and quantity of the interfaces between the modules of the system. Cohesion is a qualitative measure of the functional strength of the modules in the system (how tightly the elements of a module are bound). These two concepts are key features of the structured design methodology. However, they are also regarded universally as important measures of design.

Two other concepts that play an important role in the application of ADM are information hiding and abstract data types. These two concepts are used to enhance a design beyond what can be achieved by applying only coupling and cohesion as measures of design. Information hiding involves making visible to a module only the data that it needs in order to perform its function. The concept of abstract data types promotes the development of operations around an abstract data structure into one module. Ada supports these design concepts better than most high-level languages.

The four design concepts referenced above (coupling, cohesion, information hiding, and abstract data types) heavily influenced the development of ADM. Therefore, it is important that the designer have a good knowledge of these concepts.

1.3.3.1.6 Contribution of Ada

ADM uses concepts supported by Ada and certain Ada constructs to help determine system architecture. It also uses constructs of Ada as a PDL.

Ada concepts that support developing system architecture include packaging and tasking. Packaging is the vehicle used to achieve informational strength modules in design. Information strength is the goal of object-oriented design which is applied by ADM. Tasking is used by ADM to support concurrency and the performance and real-time constraints associated with a communications system. Specifically, tasking supports concurrent processing and synchronization which are important characteristics of communication systems. The Ada task construct itself provides for concurrent operations in the Ada environment. Task communication and synchronization between tasks (concurrent operations) is accomplished through the rendezvous mechanism in Ada.

Ada language constructs are used as a PDL at two distinct points in the application of design. First, Ada is used to express the basic framework of system design before hardware components are identified. This first Ada PDL expression of system design describes the interfaces between the subprograms, packages, and tasks within the system design as they exist prior to hardware/software partitioning. The second Ada PDL expression of the system design occurs after hardware/software partitioning. This second expression goes beyond describing the basic framework and interfaces of the system design. It expresses the functional requirements of each subprogram and package in the system architecture. Additionally, the basic requirements of each hardware component in the system architecture is described in the second Ada PDL expression.

1.3.3.2 Scope of ADM

The application of ADM produces graphic illustrations of the system design. ADM also facilitates the development of programming requirements and design documentation. This is normally within the scope of a design methodology. However, because ADM is an Ada-based methodology, there is a tendency toward using the Ada language as much as possible during design. Also, the nature of the Ada language requires the designer to function somewhat like a chief programmer during design. For example, the designer must evaluate the overall design in terms of data types, overloading of functions, etc. Any encumbrances created by improper data typing or improper overloading of functions should be eliminated before actual program development begins. ADM's use of Ada as a PDL for developing programming specifications is another example of the impact and use of the Ada language during design.

The Ada influence on ADM moves design closer to the implementation/program development phase of the traditional system development project life cycle. Considering this, ADM broadens the scope of design. However, on the other side of the project life cycle, ARM includes some tasks that have traditionally been considered within the scope of design.

1.3.3.3 Outputs

The output of ADM is an Ada Design Document. Components of this document are listed below:

a. System Chart

b. Structure Chart

c. Packages Chart

d. N^2 Chart

e. Ada PDL expression of the system

f. Traceability Matrix

g. Design Philosophy Document

h. Logical Data Structures Model

i. Jackson Data Structures.

The Ada Design Document is intended to provide the information necessary for implementing the design.

1.4 RECOMMENDATIONS FOR PERSONNEL USING AIM

As stated earlier, it was assumed that the requirements analyst(s) and the designer(s) are not the same person(s). Therefore, separate recommendations are provided below for each of these positions.

1.4.1 Requirements Analyst Recommendations

The requirements analyst supplying ARM should be familiar with the Structured Analysis and SADT methodologies. Ideally, he will have experience applying one of these methodologies. It is also recommended that the requirements analyst become sufficiently familiar with selected constructs of the Ada language to use it as the basis for writing structured English functional specifications.

Of course, it may not always be possible to assign a requirements analyst with the above qualifications to a development project. Therefore, some requirements analyst training recommendations are provided below:

a. Attend a Structured Analysis course (e.g., a week-long Yourdon Structured Analysis course or a Deltak Structured Analysis course).

b. Develop familiarity with SADT (e.g., read the SADT articles listed in Table 1.4.1-1).

c. Develop familiarity with the Ada language (e.g., read the sections of an Ada text or reference manual such as the ones listed in Table 1.4.1-1 that relate to the constructs used by ARM to express function requirements).

1.4.2 Designer Recommendations

The designer applying ADM needs a "tool bag" full of design tools. Knowledge of and/or experience working with four design methodologies is needed by the ADM designer. Additionally, he should understand two design concepts that are often associated with Ada - information hiding and abstract data types. Finally a good working knowledge of Ada is necessary in order to use Ada during design.

COURSES	BOOKS	ARTICLES
* Yourdon Structured Analysis	*** Page-Jones, Meilir. 1980. The Practical Guide to Structured Systems Design. New York, NY.: Yourdon Press.	* Softech. "The TRAIDEX SADT Model" (An extract from "Final Report: TRAIDEX Needs and Implementation Study").
* Deltak Structured Analysis	** Constantine, Larry L., and Yourdon, Edward. 1978. Structured Design. New York, NY.: Yourdon Press.	* Softech. Nov., 1976. "An Introduction to SADT Structured Analysis and Design Technique". Waltham, MA. Softech.
** Yourdon Structured Design	** Jackson, M. A. 1975. Principles of Program Design. London, NW1: Academic Press, Inc.	* Ross, Douglas T. Sept. 16, 1976. "Structured Analysis (SA): A Language for Communicating Ideas". Waltham, MA. Softech.
** Deltak Structured Design	** Myers, Glenford J. 1978. Composite/Structured Design. New York, NY.: Van Nostrand-Reinhold Company.	* Cole, George E., Jr.; Conn, Alex Paul; Yeh, Raymond T.; and Zave, Pamela. October, 1980. "Software Requirements: A Report on the State of the Art". University of Maryland.
** Software Engineering	*** Ledgard, Henry. July, 1980. Ada: An Introduction. Ada Reference Manual. New York, NY.: Springer-Verlag.	* Booch, Grady. Sept., 1981. Describing Software Design in Ada. ACM SIGPLAN NOTICES Volume 16.
** Ada Design	*** Pyle, I. C. 1981. The Ada Programming Language. London: Prentice-Hall International.	

*   Applies to ARM
** Applies to ADM
*** Applies to ARM and ADM

Table 1.4.1-1 Suggested Training Materials and Courses for ARM and ADM Users.

In short, the ADM designer should be an experienced, state-of-the-art software engineer. Realizing that it may be difficult to assign such a designer to each development project, two recommendations for training an ADM designer are provided below:

a. Attend a Yourdon Structured Design course or a software engineering course that addresses structured design, object-oriented design, and data-driven design.

b. Attend an Ada design course.

1.4.3 Sources for Study and Training

Table 1.4.1-1 is a concise list of courses, books, and articles that may be appropriate for study before applying AIM. A bibliography which lists all of the books, articles, class outlines, etc., accumulated during the study is attached and may be used to select study material for requirements analysts and designers expected to use AIM.

1.5 SYNOPSIS OF SUCCEEDING CHAPTERS

Chapter 2 describes ARM and directs the requirements analyst in its application. Each component of ARM is described, and numerous examples and guidelines are provided to help the requirements analyst apply ARM.

Chapter 3 is devoted to ADM. The purpose of this chapter is to provide a brief description of ADM and direction for the designer applying it. Again, numerous examples and guidelines are provided.

The complexity and power of Ada requires careful discretion from the Ada designer/programmer. Therefore, there is a need for a set of Ada development standards. Chapter 4 is an accumulation of Ada development standards developed during the Ada Capability Study.

Chapter 5 summarizes the methodology experiences and recommendations that evolved during the study. This includes lessons learned, recommendations for improving AIM, and recommendations for further research.

CHAPTER 2

ADA REQUIREMENTS METHODOLOGY (ARM)

2.1 PREFACE

As illustrated in Figure 1.3.3-1, ARM converts the A-specifications to an Ada Requirements Document which feeds into the design process. The purpose of converting the A-specifications to another document is twofold - (1) to restate the A-specifications as structured, organized requirements and (2) to effect an understanding of the problem which can be conveyed to the designer.

ARM develops system requirements in four parts as follows:

Part I	Functional Requirements
Part II	Non-Functional Requirements
Part III	Concurrency Requirements
Part IV	Ada Requirements Document

Part I consists of four major tasks: (1) functionally decompose the problem using DFDs, (2) develop a logical data structures model, (3) develop a data dictionary, and (4) state the functional requirements. The other three parts are single-step parts. Part II is the development of non-functional requirements (i.e., requirements relating to performance, reliability, maintainability, etc.). Part III is the development of concurrency requirements (charts are created to graphically illustrate concurrency). In Part IV, the Ada Requirements Document is compiled.

It is recommended that the parts be performed in order (as listed above). Figure 2.1-1 illustrates the order in which ARM steps are to be performed with concurrency where appropriate.

2.2 PART I - FUNCTIONAL REQUIREMENTS

Descriptions, guidelines and examples for the four steps of Part I are provided below.

2.2.1 Functional Decomposition Model

2.2.1.1 Description

The Functional Decomposition Model is a top-down breakdown of what has to be done in the system. The tool utilized to express it is a directed graph.

The major elements of the graph are boxes which represent functions or processes which must be accomplished, directed arcs which represent the flow of data or control information, and

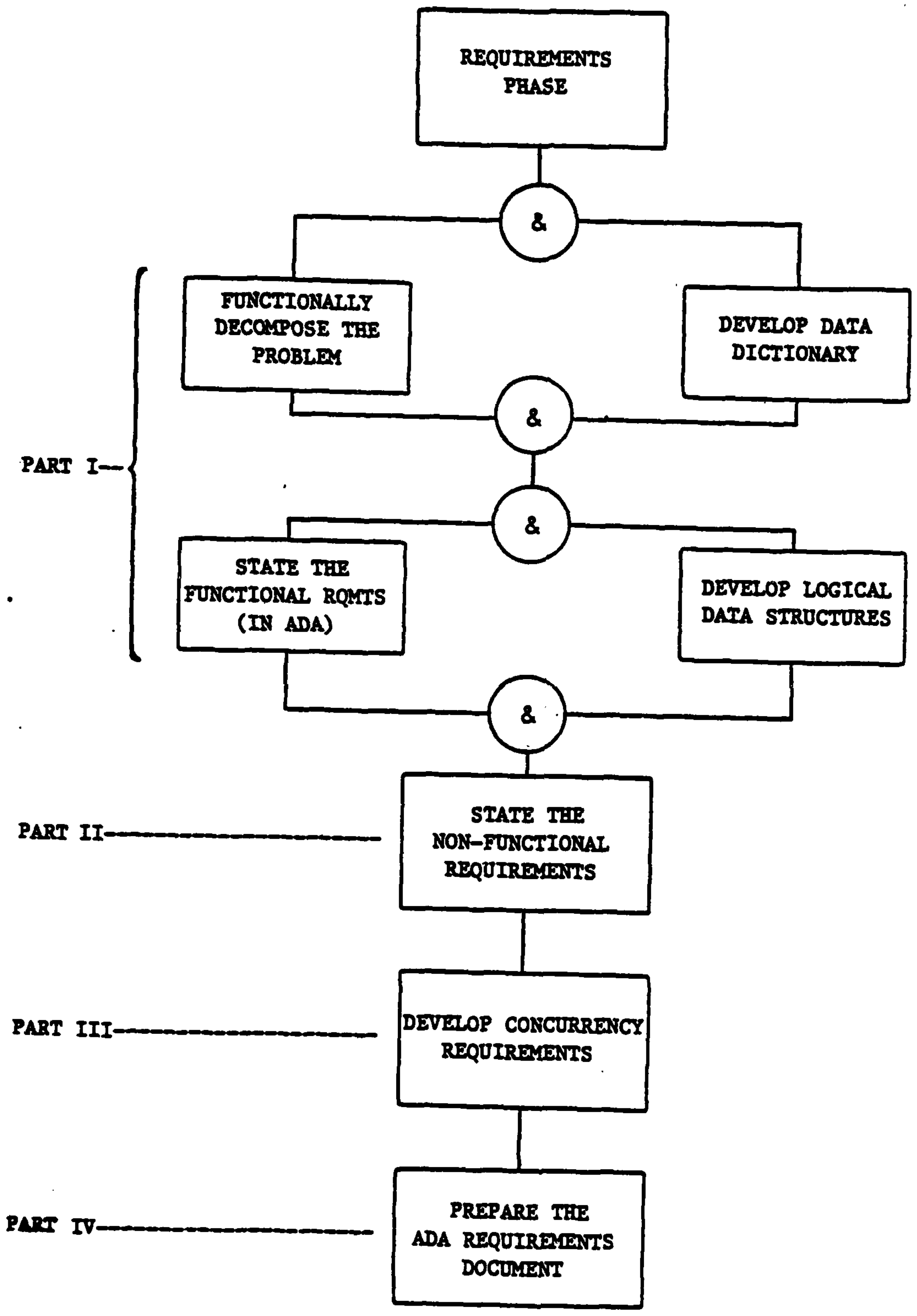

REQUIREMENTS PHASE
&
FUNCTIONALLY DECOMPOSE THE PROBLEM
DEVELOP DATA DICTIONARY
&
PART I
&
STATE THE FUNCTIONAL RQMTS (IN ADA)
DEVELOP LOGICAL DATA STRUCTURES
&
PART II
STATE THE NON-FUNCTIONAL REQUIREMENTS
PART III
DEVELOP CONCURRENCY REQUIREMENTS
PART IV
PREPARE THE ADA REQUIREMENTS DOCUMENT

Figure 2.1-1 Model for Performing Ada Requirements Methodology Tasks.

circles which may be used to represent files or temporary repositories of data. The side of the box to which an arc is connected has significance according to the following convention:

Left side - input data which will usually be consumed or modified by the function or process;

Top side - control information or data which will be utilized to control the function or process but will not normally be altered;

Right side - any and all outputs of the process or function which may then be used as input or control for any process or function, or as an output of the diagram of which this box is an element;

Bottom side - is not strictly a data flow but is utilized to reference the mechanism by which a particular function may be implemented such as a piece of hardware, or a particular software function or utility; these are not normally indicated until late requirements analysis or early design; accompanying text should clarify whether this implementation is binding.

The graph is organized into diagrams which generally have 3 to 7 boxes in them. The structure is such that the uppermost diagram has the system of interest as one box with data flow arcs indicating all inputs and outputs to it (see Figure 2.2.1-1). The next level diagram represents the functional decomposition of the system of interest into its major functions. This is illustrated by Figure 2.2.1-2. A correspondence should be maintained between the inputs and outputs to a box and the inputs and outputs to the diagram which represents the functional decomposition of that box so that no flows disappear or appear without explanation. This structuring of diagrams should be carried out in a fashion similar to the construction of SADT Actigrams or the leveling of a data flow diagram in Structured Analysis. A numbering scheme should be followed along the lines of that used in SADT.

The functional decomposition model is the major element of the requirements phase representation of the system. All other elements are directly or indirectly related to it and should be cross-referenced to individual parts of it when feasible.

2.2.1.2 Guidelines

(a) diagram A-0 shows environment in which the system of interest resides; must show all inputs and outputs for the system of interest.

(b) diagram A0 shows the whole system of interest; must reflect all inputs and outputs shown in A-0, and generally divides the system into 3-7 major logical components.

(c) each diagram reflects all inputs and outputs shown for its corresponding box in the next higher diagram; it generally divides its process into 3-7 subprocesses.

(d) elements of diagrams:

(1) boxes - represent processes or functions; should be named with a concise, meaningful statement of their function; their numbers are constructed by assigning a unique digit to each box in a diagram then appending each digit to the number for that diagram (as a general scheme numbering is assigned top to bottom, left to right)

(2) directed arcs - represent data or information; should be named with a concise, meaningful statement of their content; their numbering reflects the box they connect to and is formed by appending a prefix to a box number to indicate which side of a box they connect to, many data flows will have several numbers assiciated with them

(3) circles - represent files or temporary repositories of data; these should be given unique numbers for the entire system; files should be introduced at the level where they have an interface with more than one box

(4) small circles - may be utilized for on-diagram connectors when a data flow arc would otherwise have to cross several other data flows; they should have unique numbers to distinguish them when several occur on the same diagram

(5) mechanism arcs - short arcs connected to the bottom of boxes represent a mechanism, implementation, utility, or common function; should be named with a concise, meaningful statement of what they represent; when representing a common function a list of equivalent boxes enclosed within parentheses is an appropriate label.

(e) numbering scheme:

(1) letter portion:

A	=	activity, function, process
I	=	input
C	=	control
O	=	output
M	=	mechanism, implementation
D	=	data

(2) number portion - each digit corresponds to the respective level of the structure and indicates to which element of that level it belongs

(3) formation - each box in the graph has its own unique number; each diagram uses the number of the box that it expands (except the overview diagram which has no corresponding box and is therefore numbered A-0); each data item may be given its own unique number but will be cross-referenced to a composite number indicating the box prefixed by a use indicator for the data flows in which it occurs.

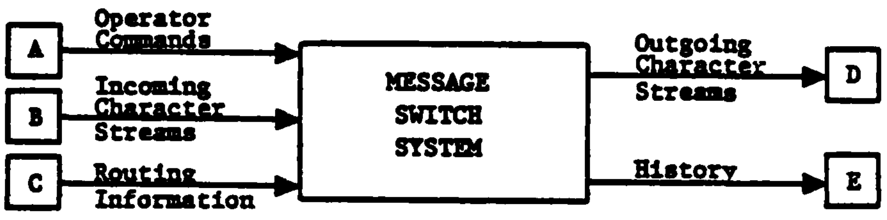


Figure 2.2.1-1 Interface/Environment System Model (Diagram A-0)

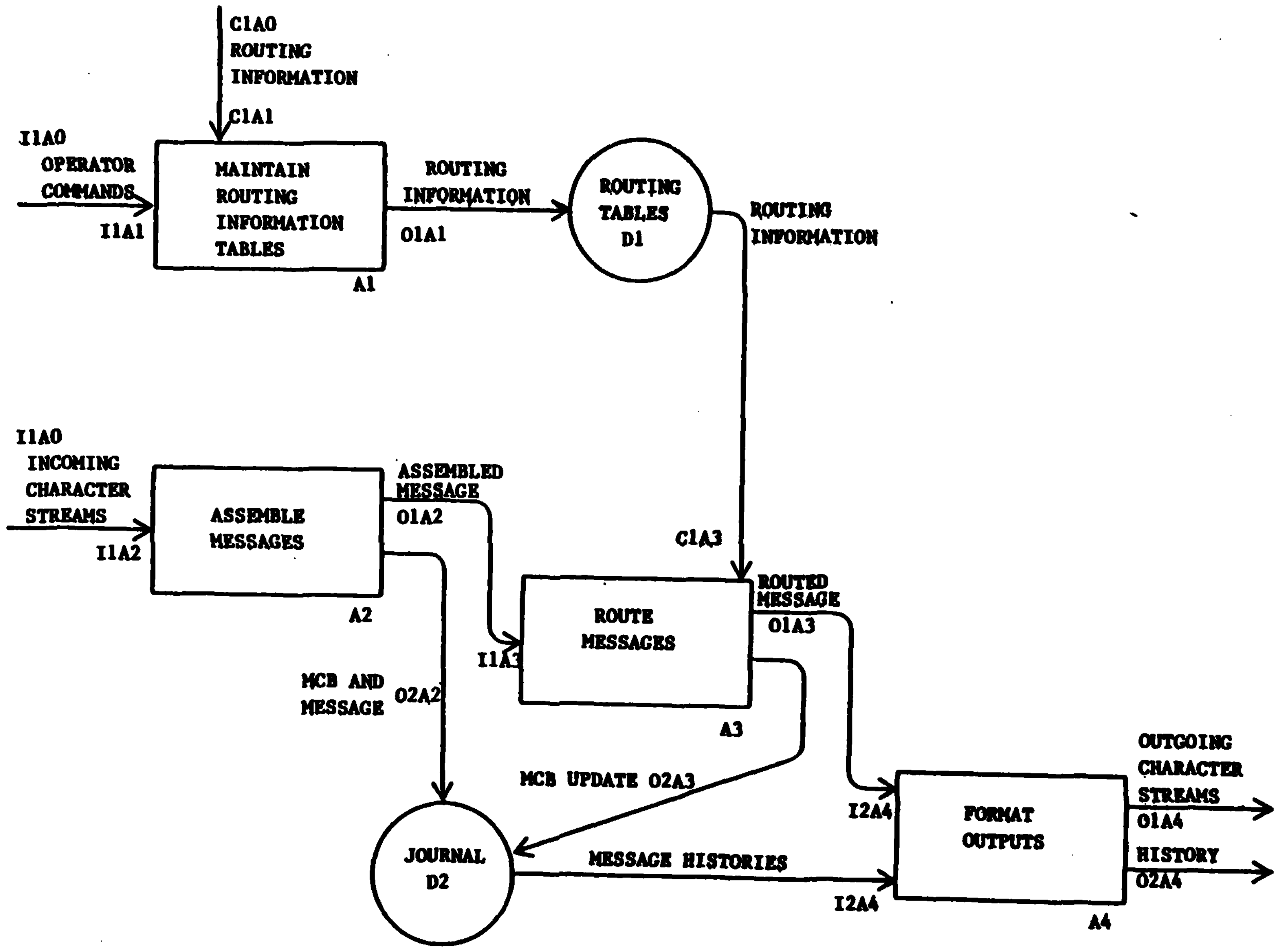


Figure 2.2.1-2 Major Functions Model (Diagram A0)

2.2.2 Data Dictionary

2.2.2.1 Description

The data dictionary is a textual listing of the items which flow along the data flow arcs of the Functional Decomposition graph. The entries should be recorded with meaningful names and cross-referenced to the respective diagrams. Emphasis should be on logical structure. The logical structure of a composite data item may be shown using the Structured Analysis data dictionary concepts of sequence, repetition, and selection.

The data dictionary lists all the logical components of the data flows that are used in the diagrams of the Functional Decomposition. It provides the information for the development of the Logical Data Structures.

2.2.2.2 Guidelines

1. Sequence: "...+..."

```
D241  Message Control Block = header info
01A24             + time of entry
                  + time of delivery
```

2. Repetition:

```
"<a>(...)"  to indicate a specific
            number of repeats
"<a|b|c|...>(...)" to indicate a
            variety of specific number
            of repeats
"<a..b>(...)" to indicate a range
            of possible number of
            repeats
default a = 1, b = infinite "<..>(...)"
```

```
D23114  Routing-Indicator = R + <6>(any-letter)
I1A321
```

3. Selection: "[...|...]"

```
        or "[a..b]" to indicate one value in a
                     range of values
D23112  Message Precedence = [critic|ECP|Flash|
I12A231   Immediate|Priority|Routine]
```

4. Option: "(...)"

5. Items to record for each entry:

   a. Data Reference Number: D12...

   b. Name: SOME-NAME

   c. Composition: = ... (when applicable)

   d. Values and meanings: (when applicable)

e. Related Data Flows: I1A241, O1A243

6. Comments may be denoted by enclosing them within asterisks.

```
1.    ROUTING-INDICATOR = RI-CODE + TT-CODE

      RI-CODE = <1..7> (ALPHABETIC CHARACTERS)
      TT-CODE = [MESSAGE SWITCH|
                 DIRECT TERMINAL EQUIPMENT|
                 AUTODIN BACKBONE|
                 AUTODIN NON-BACKBONE]

2.    TRUNK-TYPE = TT-CODE + COMMUNITY-CODE

      TT-CODE = *SEE TT-CODE ABOVE*
      COMMUNITY-CODE = [Y|R|U]
                        *Y = INTELLIGENCE*
                        *R = STRATEGIC*
                        *U = TACTICAL*

3.    LINE-NUMBER = LN-CODE + LINE-STATUS + TT-CODE

      LN-CODE = [1|2|3|...|50]
      LINE-STATUS = IDLE-CODE + BUSY-CODE + AVAILABLE-CODE

           IDLE-CODE = [1|0] *1 = TRUE, 0 = FALSE*
           BUSY-CODE = [1|0] *1 = TRUE, 0 = FALSE*
           AVAILABLE-CODE = [1|0] *1 = TRUE, 0 = FALSE*

      TT-CODE = *SEE TT-CODE ABOVE*
```

Figure 2.2.2-1 Data Dictionary Entries for Three Logical Data Structures

2.2.3 Logical Data Structures

2.2.3.1 Description

A logical data structure model is a graphic representation of the logical data components of the new system and their relationships to each other, as perceived by the requirements analyst. The word "logical" implies that the structure is not physical. However, the logical data structures model developed in the requirements phase is intended to be a primary input to the designer when he/she determines the physical data structures for the new system.

A logical data structures model is composed of the following:

a. Logical data components
b. Keys for accessing each component
c. Relationships (in the form of pointers) between the components.

The model, along with the data dictionary, which describes the contents of each logical data component, provides the designer an excellent view of the data structures required by the system. With this information, the designer must determine the physical data structure(s) (e.g. hierarchical data base, random access file, FIFO queue, stack, etc.) needed for the new system.

2.2.3.2 Guidelines

a. Identify all the file accesses in the existing system and group them into input and output categories.

b. Identify the key for each access.

c. Analyze each input and output access in terms of the data flow that is actually required versus the present data flow.

d. Develop logical components for the required data flows.

e. Analyze each logical component in terms of each relationship to the key and separate unrelated data elements (into separate logical components).

f. Separate repeating groups into new logical components.

g. Determine relationships (pointers) between all the logical components developed.

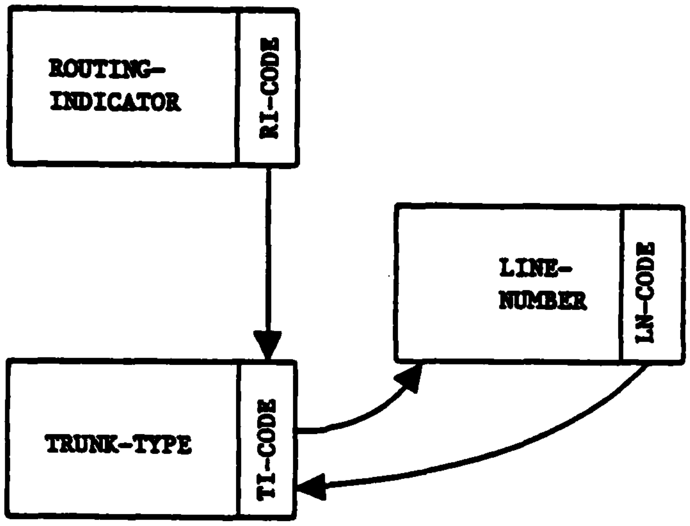
h. Develop a model of the logical data structures.

Figure 2.2.3-1 illustrates a possible physical file structure for an existing message switch system that is to be replaced or redesigned. In this example, all data elements are combined into one data structure. Figure 2.2.3-2 illustrates how a requirements analyst might represent such a file structure logically during the requirements phase of a project to redesign the message switch system.

In order to come up with the logical data structure in Figure 2.2.3-2, the requirements analyst would analyze the data flow requirements for each access to existing file structures. Based upon the analysis of the current file structure, new logical data structures (consisting of only the data elements required for each access) would be developed. Then these initial logical data structures would be evaluated, refined, and modeled.

ROUTING INDICATOR	TRUNK TYPE	LINE NO.	LINE STATUS
RI(1)	TT(1)	LN(1)	LS(1)
RI(1)	TT(1)	LN(2)	LS(2)
RI(2)	TT(2)	LN(3)	LS(3)
RI(3)	TT(1)	LN(1)	LS(1)
RT(3)	TT(1)	LN(2)	LS(2)
RI(4)	TT(3)	LN(4)	LS(4)
RI(4)	TT(4)	LN(5)	LS(5)
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
.	.	.	.
RI(n)	TT(n)	LN(n)	LS(n)

Figure 2.2.3-1 Possible physical file structure for an existing message switch system



Explanation: Data structure components are represented by boxes which are divided into two parts - (1) leftmost part represents the logical component and (2) rightmost part represents the key of each logical component.

Relationships (pointers) between data structures are represented by directed arcs.

*Logical Components - Routing-Indicator (RI-code,TT-code)
- Trunk-Type (TT-code,LN-code)
- Line-Number (LN-code,Line-status,TT-code)

*Keys are underlined for each logical component.

Figure 2.2.3-2 A logical data structures model of the physical file structure illustrated by Figure 2.2.3-1.

2.2.4 Ada Functional Requirements

2.2.4.1 Description

The Ada Functional Requirements use Ada language constructs along with entry names from the Data Dictionary to express structured descriptions of the functions to be accomplished by processes that appear on the lowest level diagrams of the Functional Decomposition. The purpose for using Ada notation to express requirements is to provide a standard medium for communication that will provide a clear, concise, and non-ambiguous statement of requirements.

2.2.4.2 Guidelines

a. Use the following permissible Ada constructs:

1) if-statement
2) case-statement
3) loop-statement
4) assignment-statement
5) code-statement
6) expression
7) relation
8) exit-statement
9) procedure calls for pre-defined procedures
10) raise-statement
11) exception-handler

b. Use Data Dictionary entry names with above Ada constructs to express requirements.

c. When above constructs are insufficient to express requirements, use a disciplined English statement preceded by "-->".

d. Do not use a declaration; all items should appear in the Data Dictionary.

e. For explanative comments use a comment in an Ada format (i.e., preceded by "--").

See Figure 1.3.2-1 for an example of Ada functional requirements.

2.3 PART II - NON-FUNCTIONAL REQUIREMENTS

2.3.1 Description

Oftentimes the functional requirements of a system receive so much attention that the non-functional requirements are neglected. However, certain non-functional requirements can significantly impact the design of a system. Some of the types of non-functional requirements to be considered and stated in the requirements document are (1) performance, (2) reliability, (3) security, and (4) operating constraints. An excellent description of non-functional requirements is found in the paper, "SOFTWARE REQUIREMENTS: A REPORT ON THE STATE OF THE ART".

2.3.2 Guidelines

a. Identify all the non-functional requirements of the system that could impact design.

b. List each non-functional requirement identified in a separate section of the requirements documents.

c. Use the article, "SOFTWARE REQUIREMENTS: A REPORT ON THE STATE OF THE ART" pp 22-30, Yeh, Zave, Conn, & Cole. This article is Technical Report-949 and was published by the University of Maryland in October 1980.

2.4 PART III - CONCURRENCY REQUIREMENTS

2.4.1 Description

When concurrency is required in the specification it should be separated out and specifically stated in this portion of the requirements document. Any related requirements that will assist the designer in making decisions about how the concurrency would best be implemented should also be included in this portion even though they may also appear elsewhere. This may include such information as how many items must be processed simultaneously, or the rate at which items may enter or depart the system, or minimum and maximum response time required of specific processes. This data will greatly assist a designer when he is determining the best mechanism to use for concurrency in this particular case.

When the concurrent operations required in the system are non-trivial, then a concurrency model is introduced. The tool utilized to express it is a directed graph similar to the R-net used with Software Requirements Engineering Methodology (SREM). This is composed of boxes to represent the processes or functions, directed arcs which indicate flow, and circles with special symbols at junctions.

2.4.2 Guidelines

a. When feasible function names should correspond to names used in the functional decomposition model.

b. Only include processes of interest to the concurrency viewpoint in the concurrent model.

c. Segregate the concurrency model from the functional decomposition model.

d. The special symbols to be used in junction circles are the following:

& - "and", these circles will have one input arc and multiple output arcs; they represent initiation of concurrent execution flows.

| - "or", these circles will have one input arc and multiple output arcs; they represent a selection among non-concurrent flows.

+ - "reunion", these circles will have multiple input arcs and one output arc; they will represent the convergence of differing execution flows and in the case of concurrent input flows indicate that all flows must reach this point before the resulting flow may start.

letters - "exclusicn", these circles will occur in pairs designated by a specific letter for each pair; the first of the pair will have multiple inputs and will signify that only one flow can proceed through the following processes; the second of the pair will have multiple cutputs signifying that the appropriate flow can proceed and that another flow may proceed through the enclosed processes.

e. When there is more than one area in which concurrency is an issue use separate concurrency models to express them.

f. If a concurrency model is tco complex then leveling similar to that utilized in the functional decomp^sition model may be used to structure the concurrency model into understandable pieces.

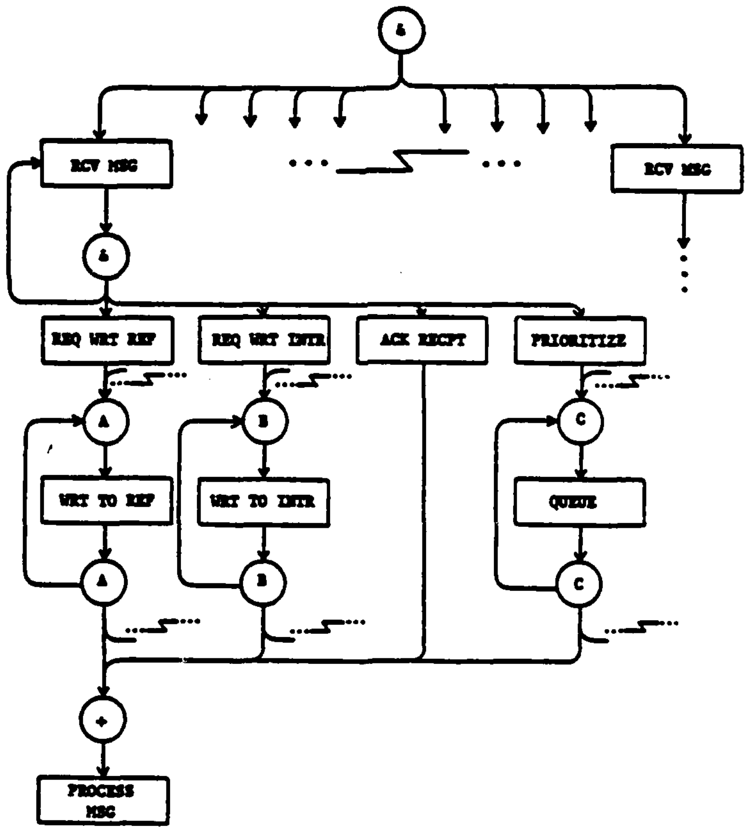


Figure 2.4.3-1 Concurrency model for a portion of a message switch.

2.5 PART IV - ADA REQUIREMENTS DOCUMENT

2.5.1 Description

The Ada Requirements Document is primarily the compilation of the outputs from each previous phase of the requirements phase. It is the mechanism through which the requirements analyst's understanding of the problem is to be conveyed to the designer.

2.5.2 Guidelines

a. Write a brief description of the problem (system) that was studied during the requirements phase. Give reasons why the present system (if there is one) needs to be changed.

b. Organize the Ada Requirements Document according to the outline below:

I. SYSTEM OVERVIEW

A. Brief Description of System Studied
B. Reasons for Study

II. FUNCTIONAL SPECIFICATIONS

A. Environment Data Flow Diagram (A-0)
B. Data Flow Diagram of All Major Functions of the System (A-0)
C. Data Flow Diagrams and Ada Requirements Specifications for Each Major Function of the System
D. Data Dictionary
E. Logical Data Structures Model

III. NON-FUNCTIONAL REQUIREMENTS

IV. CONCURRENCY REQUIREMENTS - CHARTS

CHAPTER 3

ADA DESIGN METHODOLOGY (ADM)

3.1 PREFACE

Within the system development life cycle, the design phase produces a detailed approach to solving the problem characterized by the requirements phase. The design phase itself consists of three parts: architectural design, detailed design, and compilation of the Ada Design Document. Given the Ada Requirements Document, which is the output of the requirements phase, as input, the architectural design step produces the following items to be used as input to the detailed design phase:

a. a graphical representation of the system structure (modified structure chart)

b. system flowchart

c. N^2 chart

d. packages chart

e. data structure charts

f. Ada unit specifications.

Furthermore, the architectural design must be correct (every requirement listed in the requirements document is satisfied), and good (it should be maintainable, efficient, etc.). The former criteria can be approached, in part, by using a traceability matrix that maps requirements into modules in the architectural design. The standard metrics for goodness are coupling and cohesion.

The steps to be followed during architectural design are described in detail in subsequent sections. The order of the architectural design steps is:

a. Develop data structures for all the files, data bases, and intermediate data repositories required by the system

b. Perform an object-oriented design pre-analysis

c. Develop concurrency requirements for the system

d. Generate a structure chart

e. Validate the goodness of design by applying: an analysis of coupling, an analysis of cohesion, and a post-analysis of the packaging

f. Validate the correctness of design by constructing a traceability matrix to link each requirement with the Ada design unit responsible for its implementation

g. Create an N^2 chart based on the data flow interfaces between the functions on the structure chart

h. Perform a preliminary design review and iterate any combination of the steps above as required

i. Develop Ada unit specifications for each Ada design unit of the structure chart

j. Perform hardware/software partitioning of the system.

Detail design produces an Ada representation of the architectural design. Once the Ada representation of the system is completed, the design is verified and traced back to the system requirements (Ada Requirements Document).

The steps of detail design are described in subsequent sections. The order of the detail design steps is:

a. Express the system design in Ada PDL

b. Perform a final design review which consists of the following:

    1. Design Walk-Thru
    2. Preprogramming Ada Evaluation
    3. Requirements-To-Design Traceability
    4. Design Philosophy Review

Compilation of the Ada Design Document is the third and final part of design. This phase consists of organizing all the design outputs into a formal document.

The sections that follow give guidance for applying the architectural and detail design steps listed above. However, the guidance provided is somewhat general, and before applying ADM, the designer should read the description of ADM and recommendations for personnel applying ADM in Chapter 1 (INTRODUCTION TO ADM).

3.2 PART I - ARCHITECTURAL DESIGN

3.2.1 Data Structure Chart

3.2.1.1 Description

All data structures of the system will be described using the Jackson structure diagrams. The graphical presentation of a data structure diagram may very well provide a basis for subsequent packaging thus emphasis should be placed on the precise documentation of data structures using the Jackson structure diagram.

The Jackson method recognizes three composite types when representing data structures:

a. Sequence structure

b. Selection structure

c. Iteration structure

and one elementary component (no further decomposition possible).

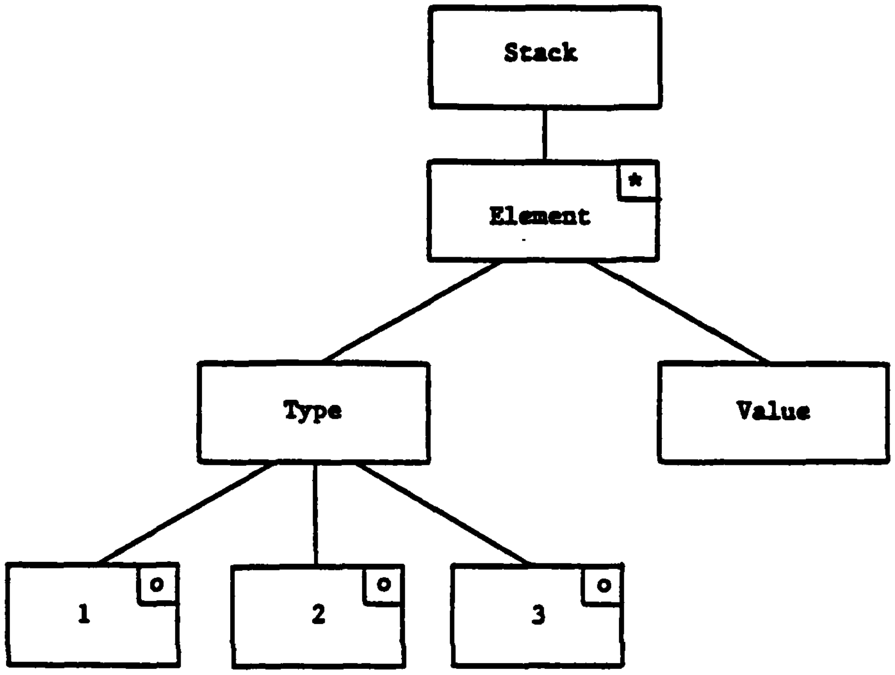
Elementary components will be at the leaf nodes of the structure diagram and composite components will be at non-leaf nodes. Selection is represented by the small circle (o) in the upper right corner of a box and iteration by an asterisk (*) in the upper right corner. Consider Figure 3.2.1-1 which depicts a stack having elements consisting of two parts - a type (1, 2, or 3) and an integer value:

As an extension to the Jackson approach we introduce the symbol plus (+) to indicate one or more occurrences of a component. Thus, in the prior example, if the semantics did _not_ permit an empty stack we would have used the plus sign (see Figure 3.2.1-2) instead of using an asterisk.

3.2.1.2 Guidelines

Each logical data structure developed from the requirements phase should be re-examined and the appropriate Jackson structure diagrams generated. (Normally one structure diagram for each of the significant logical data structures.) When appropriate (e.g., a class of similar data structures), several data structures may be characterized in one diagram. Also, as additional data structures, files or databases are introduced in the design phase, the designer should modify the logical data structures model produced during the requirements phase and describe each new structure with an appropriate Jackson structure diagram.

Jackson Figures 3.2.1-3, 3.2.1-4, and 3.2.1-5 illustrate the application of this approach with a logical data structure from a requirements document, a file description and a relational database description respectively.



Elementary components: 1, 2, 3, VALUE
Sequence: TYPE followed by VALUE
Selection: 1 or 2 or 3 under TYPE
Iteration: ELEMENT occurs 0 or more times under STACK

Figure 3.2.1-1 Examples of the Jackson symbolism.

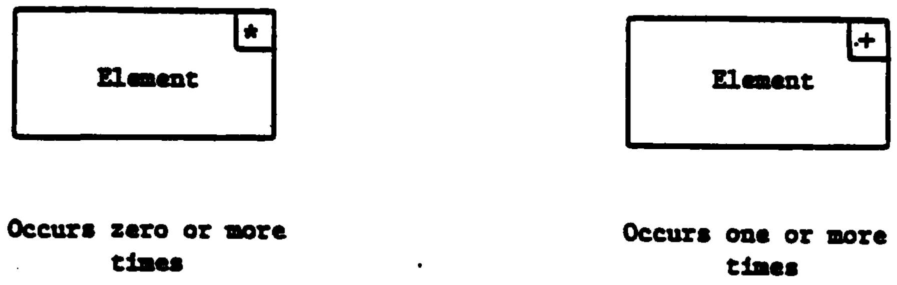


Figure 3.2.1-2 Extension to symbolism to incorporate an iteration with at least one occurrence.

Consider the following data structure (logical) representation emanating from the ARM document:

ROUTING-INDICATOR = RI-CODE + TT-CODE
RI-CODE = <1..7>(ALPHABETIC CHARACTERS)
TT-CODE = [MESSAGE SWITCH | DIRECT TERMINAL EQUIPMENT | AUTODIN BACKBONE | AUTODIN NON-BACKBONE]

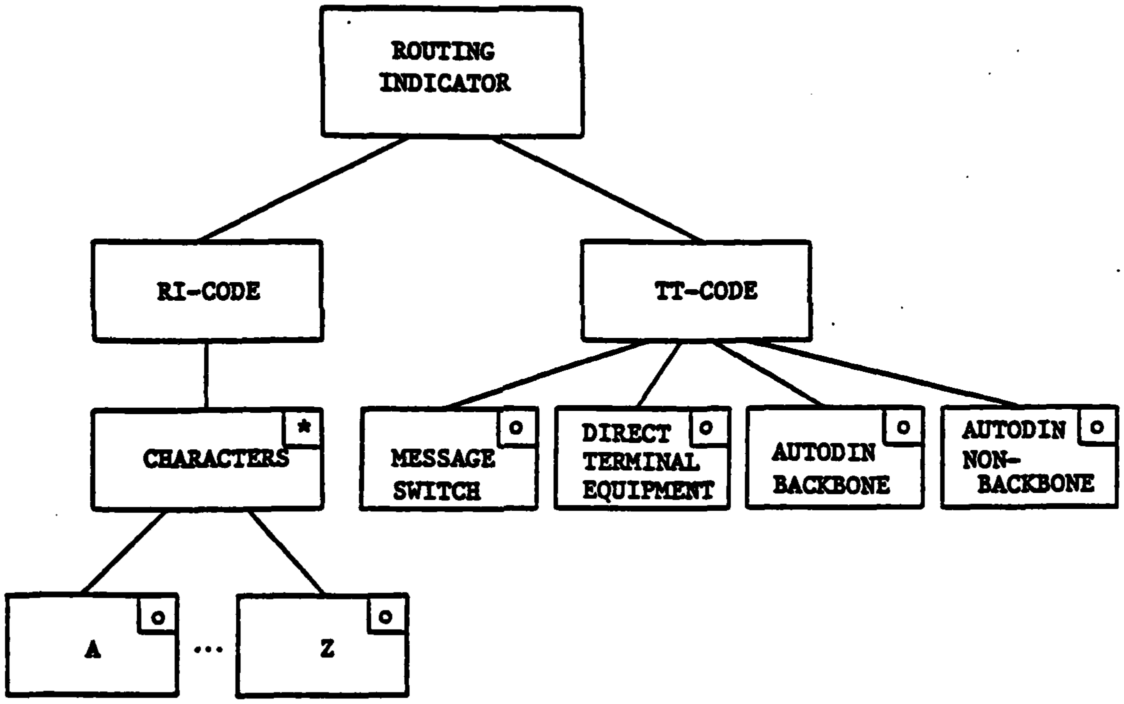


Figure 3.2.1-3 Transformation from requirements logical structure to Jackson structure diagram.

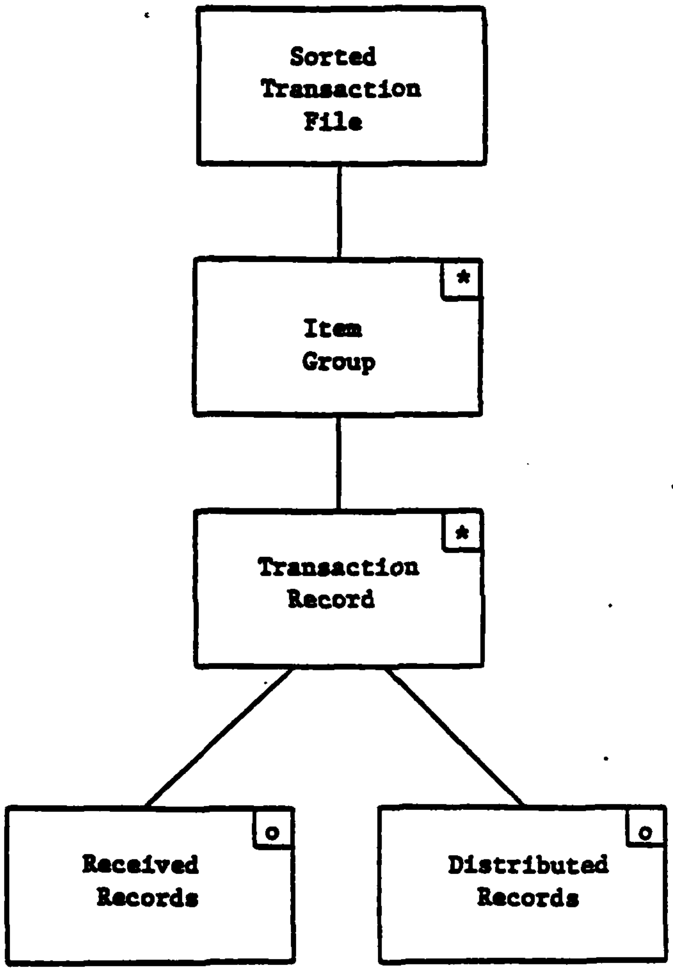


Figure 3.2.1-4 Representation of a logical file.

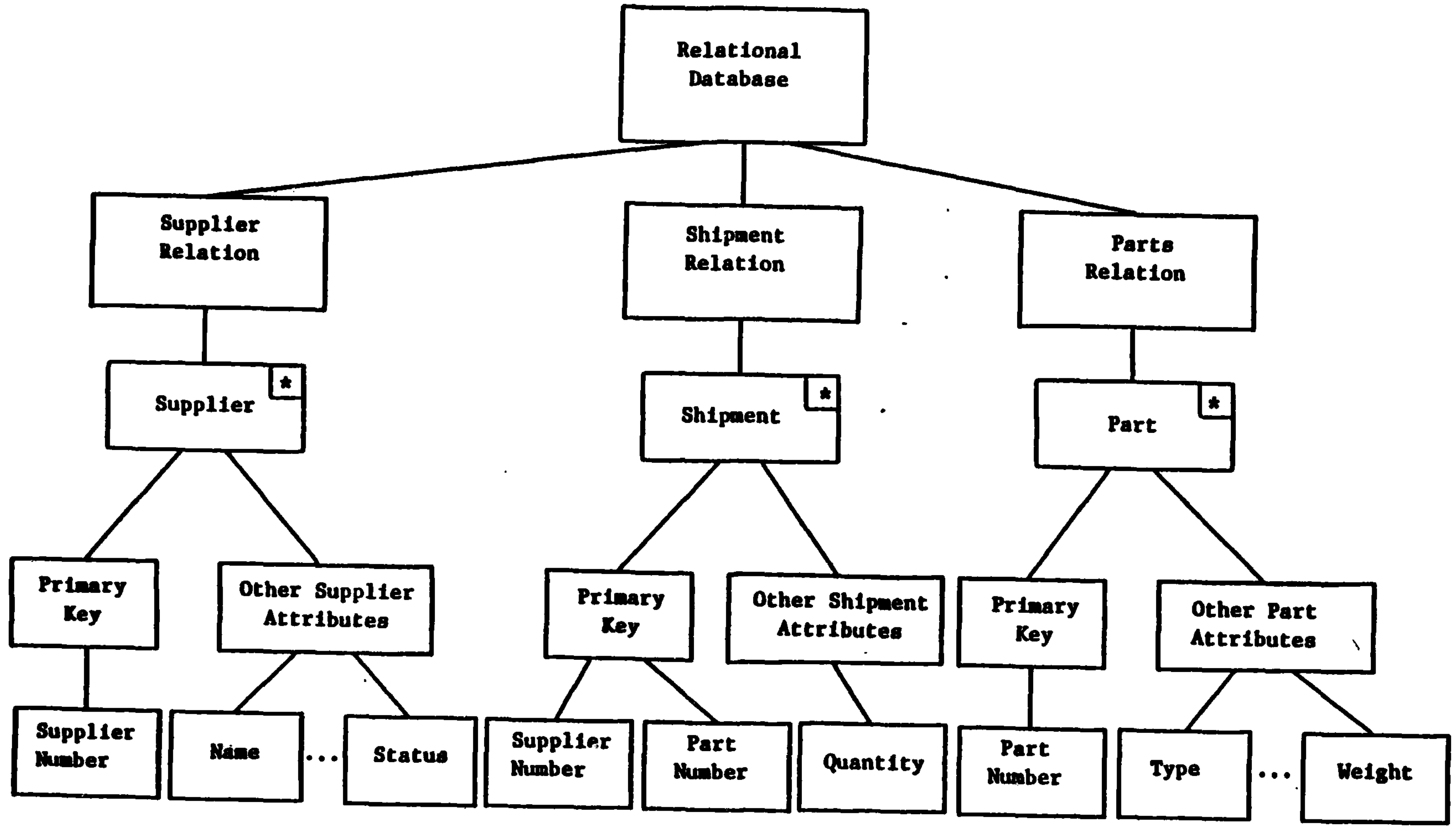


Figure 3.2.1-5 Database representation using Jackson structure chart.

3.2.2 Pre-Analysis Utilizing Object-Oriented Design

3.2.2.1 Description

Information hiding and abstract data types, both supported by Ada, are powerful design concepts used to achieve a high degree of module independence and, hence, maintainability. Because of the significance of this concept, this methodology employs two phases to identify appropriate abstract types or informational strength modules (packages). The first phase is a pre-analysis of the requirements and is object oriented. The second phase is a post-analysis of the functional decomposition and attempts to identify informational strength modules. While the pre-analysis does not drive the functional decomposition it certainly can influence the decomposition approach and provide checks and balances for the design decisions that are made (i.e., if a functional decomposition is made that potentially violates the object oriented packaging, that particular decision should be reviewed and justified). In that sense, the object oriented pre-analysis is used as a guideline or map for the post-analysis steps that are applied to each iteration of the functional decomposition. The pre-analysis follows the approach outlined by Grady Booch.

3.2.2.2 Guidelines

As stated above, this step (pre-analysis) is object-oriented. An object is an identifiable entity or item within the system which possesses attributes (characteristics). Generally objects are data structures (e.g. files, data bases, stacks, queues, etc.) or hardware components (e.g. screens, keyboards, ports, etc.) around which operations are performed. For example, a payroll master file might be an object in a payroll system. Attributes associated with a file include file name, size, and organization. Operations around a file include open, close, read, write, and update.

The steps of pre-analysis are provided below in the order that they are to be performed.

a. Develop an informal strategy for abstracting the problem solution environment.

b. Formalize the strategy.

   1. Define the objects of the system using a Jackson-like structure model.
   2. Define the attributes of the objects.
   3. Define the operations on each object in the system.
   4. In a bottom-up fashion collect common operations around objects in the system and form packages.

c. Evaluate critical operations defined in the requirements. For example, if a performance constraint in requirements is critical, the designer might want to prototype one or more of the key packages. A key package is intended to mean a package in which critical operations are encapsulated.

The identification of potential packages early in design (during the first step) provides flexibility for independent development of major components (packages) of the system. Of course, this is a deviation from top-down development.

3.2.2.3 Example

Consider the problem of developing a page editor for a small, personal computer. The system has a diskette used to hold the file to be edited; a monitor for displaying a page of the file to be edited; a keyboard for inputting characters, single character commands, and character strings for updating the file page; and, of course, the processor. A formal requirements document would enumerate the different editing operations and real-time constraints for the proposed system.

The steps of object-oriented design as applied to these requirements are depicted in Figures 3.2.2-1, 3.2.2-2, 3.2.2-3, and 3.2.2-4.

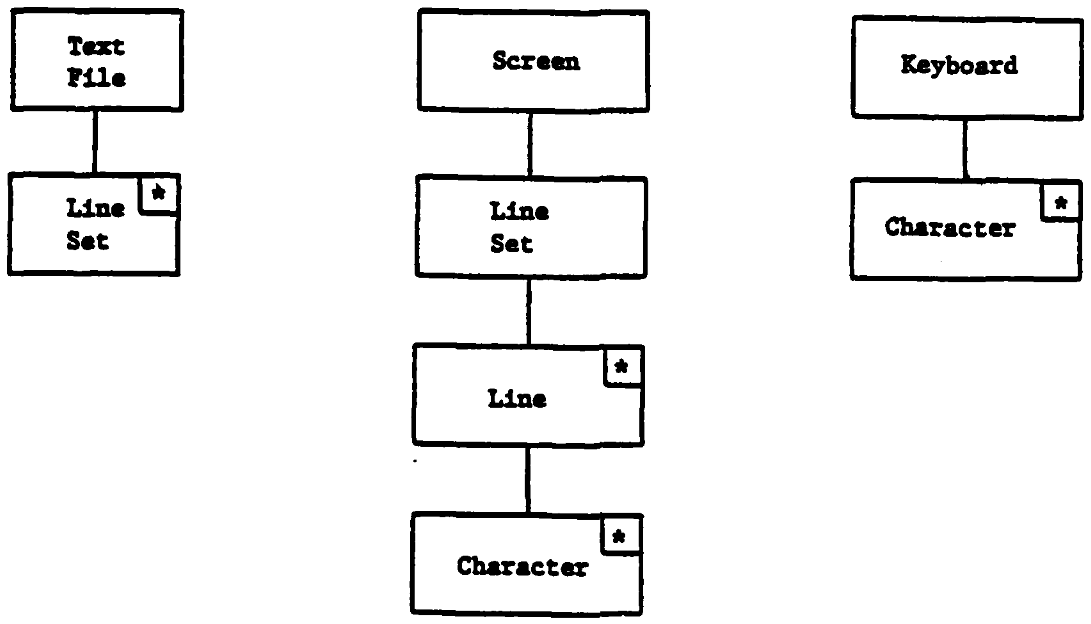


Figure 3.2.2-1 Define objects of system using Jackson symbology.

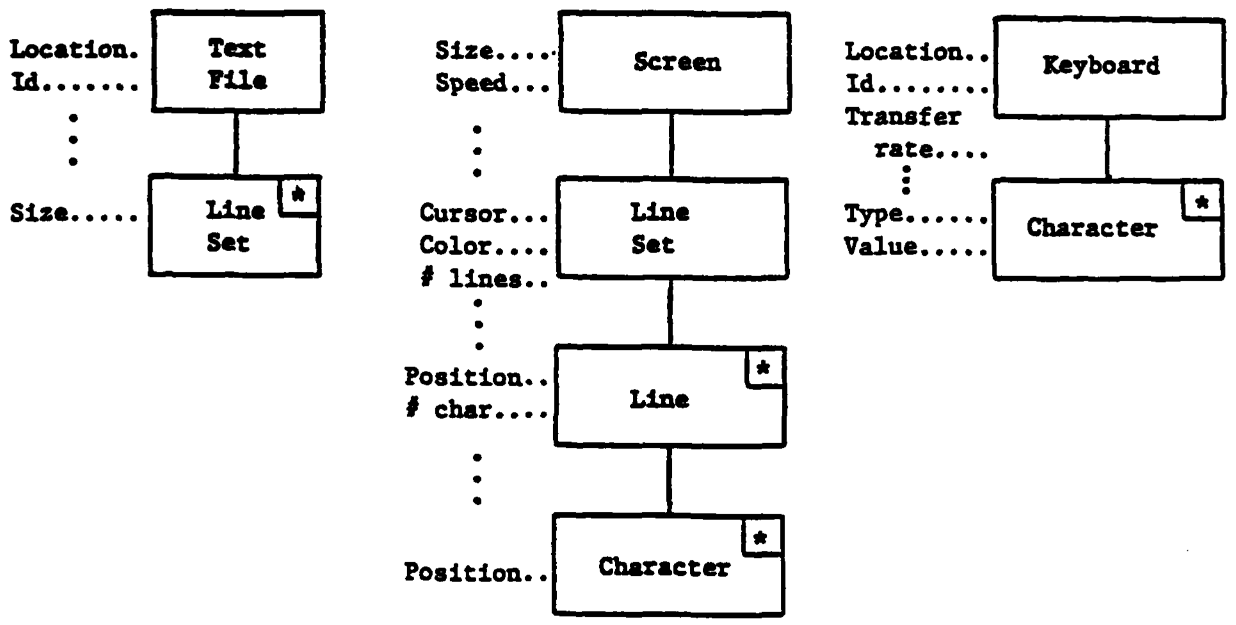


Figure 3.2.2-2 Define Attributes of the Objects.

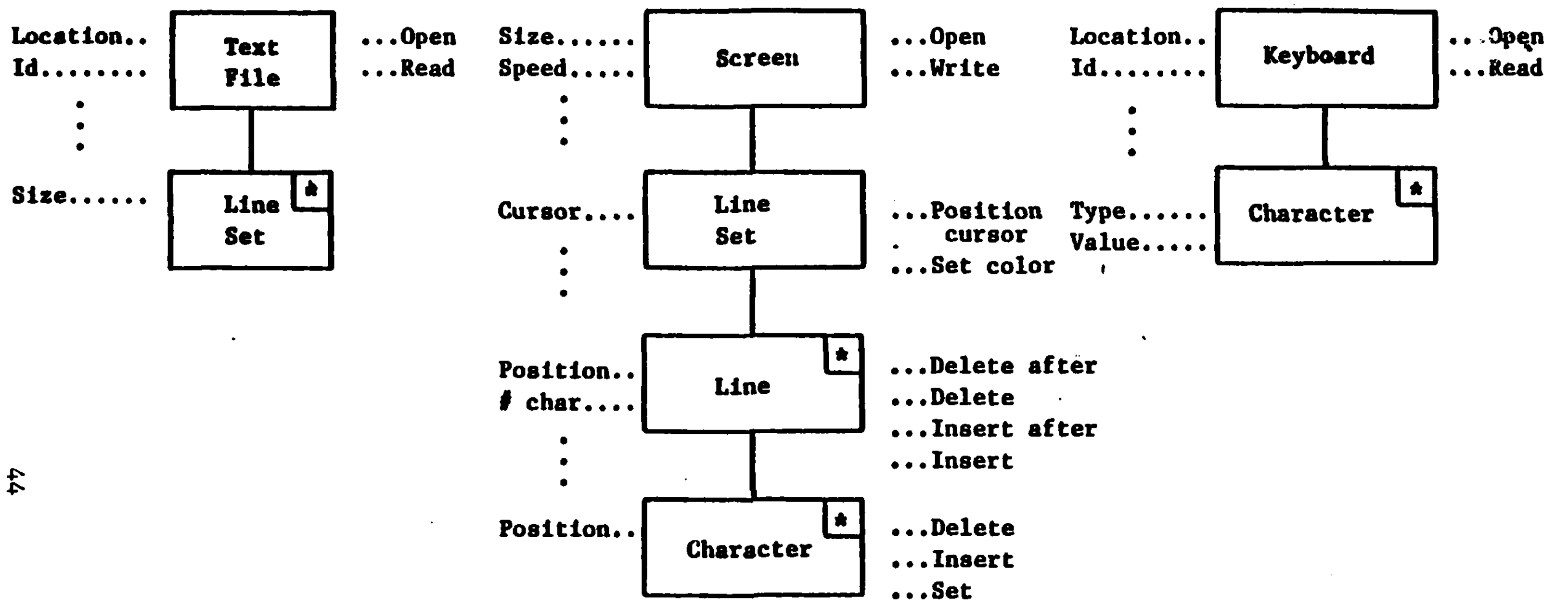


Figure 3.2.2-3 Attributes and operations assigned.

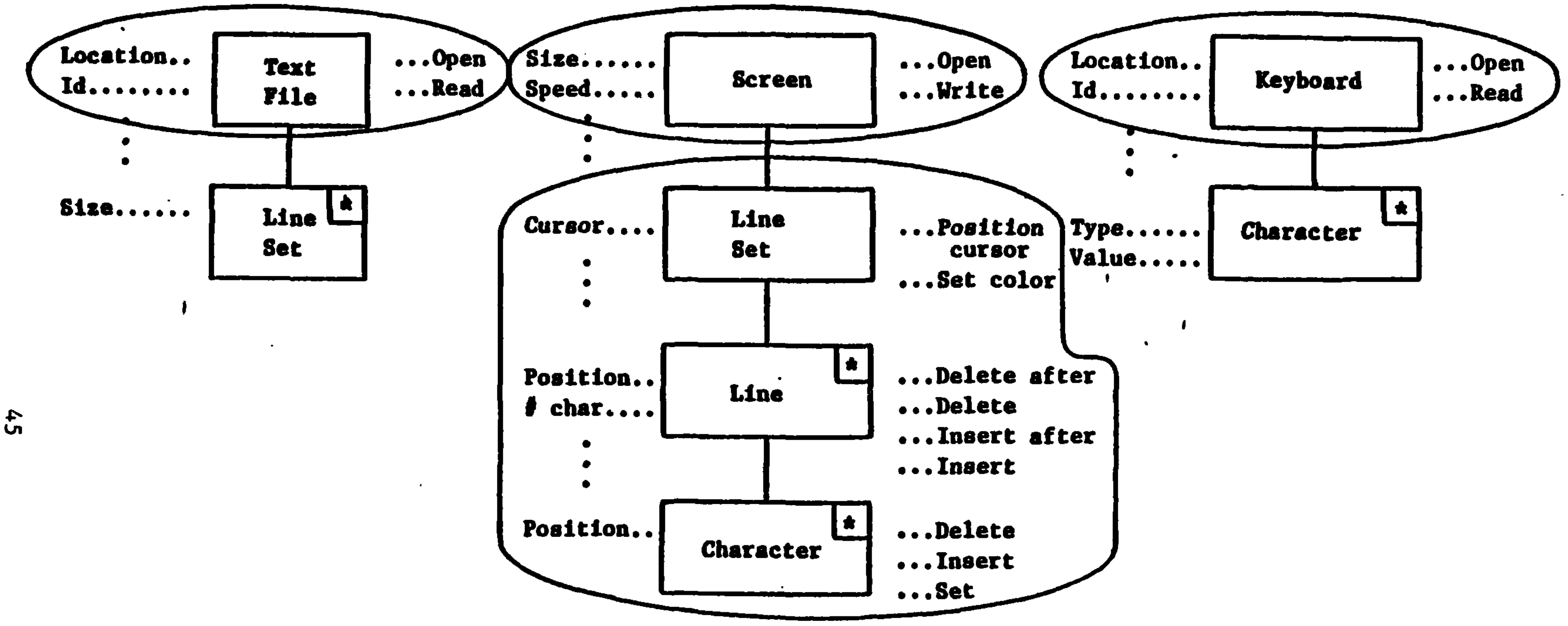


Figure 3.2.2-4 Objects identified.

3.2.3 Concurrency

3.2.3.1 Description

Concurrency is reflected in two ways in the architectural design phase:

a. In the structure chart

b. In the concurrency chart.

The structure chart describes the decomposition or structure of the system including the various types of tasks and interprocess communication. Parallelograms indicate tasks and solid/dashed lines indicate communication with tasks. The concurrency chart shows flow of control and synchronization constraints. The latter chart should be developed as the next step after the pre-analysis object-oriented design.

This final document uses the same symbology developed in the requirements phase and should be used to expand the concurrency chart by adding concurrency developed in the design phase with that developed in the requirements phase. Concurrency in the design phase may be introduced to enhance program understanding, simplify the design, or respond to certain requirement constraints. In the requirements phase, concurrency was introduced only when it modeled the problem environment (i.e., only when it was itself a constraint of the system).

3.2.3.2 Guidelines

The sequence in which the data describing concurrency is generated is important and should be as follows:

a. Determine concurrency that may be added to that already described in the requirements phase. This "design" concurrency is introduced in the following circumstances:

   1. When the design will be clearer by its introduction
   2. When it is essential to meeting a performance requirement of the system.

b. Augment the concurrency charts developed in the requirements phase as necessary to introduce the "design" concurrency.

c. From the new concurrency charts, tasks will be identified that will subsequently appear as parallelograms in the system structure chart.

d. Determine task activation requirements to be shown by dashed lines in the system structure chart.

Guidelines for development of the structure chart are given in Section 3.2.4.

Refer to the requirements section for an example of the construction of a concurrency chart.

3.2.4 System Structure Chart

3.2.4.1 Description

A structure chart illustrates the functional hierarchy and interfaces (flow of data) between the functions of the system. It is initially developed from the DFDs produced during the requirements phase and iteratively modified until the optimum system structure is achieved. The purpose of the structure chart is to be used as a tool for developing a solution to the problem studied during the requirements phase.

Once the first-cut structure chart is completed, the structure chart serves as a worksheet for iteratively studying and modifying the system structure. The goal is to modify the first-cut structure chart until optimum coupling, cohesion, packaging, and information hiding is achieved.

In addition to the traditional modules that are represented on the hierarchy chart by rectangular boxes, it is recommended that tasks (parallel execution processes) be represented by parallelograms. These parallelograms are to be incorporated in the hierarchy or structure charts parallel to the invoking module. Using this notation we can easily represent the tasks that may be executed in parallel.

3.2.4.2 Guidelines

A variety of tools have been used to produce a hierarchical structure of the computing system including functional decomposition, information hiding, Jackson methodology and structured design methodology among others. These approaches have been successfully documented on a variety of problem types. Each approach or method has individual advantages and disadvantages. ADM's approach is that of a toolbag, that is, utilize the approach that appears most useful for the problem at hand. However, rather than just reach into the toolbag, the designer should first attempt a decomposition based on the Yourdan/Constantine approach of structured design that is driven by a data flow diagram. This will permit the utilization of objects resulting from the requirements phase and provide a continuous approach across life cycle phases. Thus, the guidelines below will direct the major attention to the structured design approach with emphasis directed to the other approaches in a secondary manner.

There are two major strategies for deriving a structure chart during design: (1) transform analysis and (2) transaction analysis. The steps for implementing each of these strategies is provided below:

a. Transform Analysis steps

1. Step 1 - Develop an expanded data flow diagram (DFD) from all the lowest level diagrams produced

during the requirements phase. Simply connect all the primitive functions together to form an entire model of the system to be developed.

2. Step 2 - Identify the afferent, efferent, and central transform branches of the system. Determine the afferent and efferent branches of the system first. The remainder of the system is considered as the central transform.

The afferent branch is determined by identifying the afferent data elements. Afferent data elements are those data flows that are the last representations of input in a DFD. They are the data elements that are last in the input stream(s) of the DFD. An afferent data element is the output of the last transformation on input.

The efferent branch is determined by identifying the data flow that represents the first phase of output. The efferent data elements are those elements at the beginning of each output stream of the DFD.

Draw arcs between the afferent data elements and the transforms into which they flow. Also, draw an arc between each efferent data element and the transform that outputs it.

An example of a DFD that has been divided into afferent, efferent, and central transform branches is provided by Figure 3.2.4-2.

Step 3 - Do first level factoring. This involves identifying the subproblems of the system and forming the first two levels of the structure chart. Some subproblems are directly related to the afferent and efferent data elements and the afferent and efferent streams within the DFD. Other subproblems come from the central transform branch of the system. A possible first-cut factoring for the diagram in Figure 3.2.4-2 is illustrated by Figure 3.2.4-3.

Step 4 - Factor the afferent, efferent, and central transform branches established in Step 3. Progressively decompose each subproblem until you have decomposed it to its most primitive level. Figures 3.2.4-4, 3.2.4-5, and 3.2.4-6 illustrate factoring of the afferent, efferent, and central transform branches of the system subproblems illustrated in Figure 3.2.4-3.

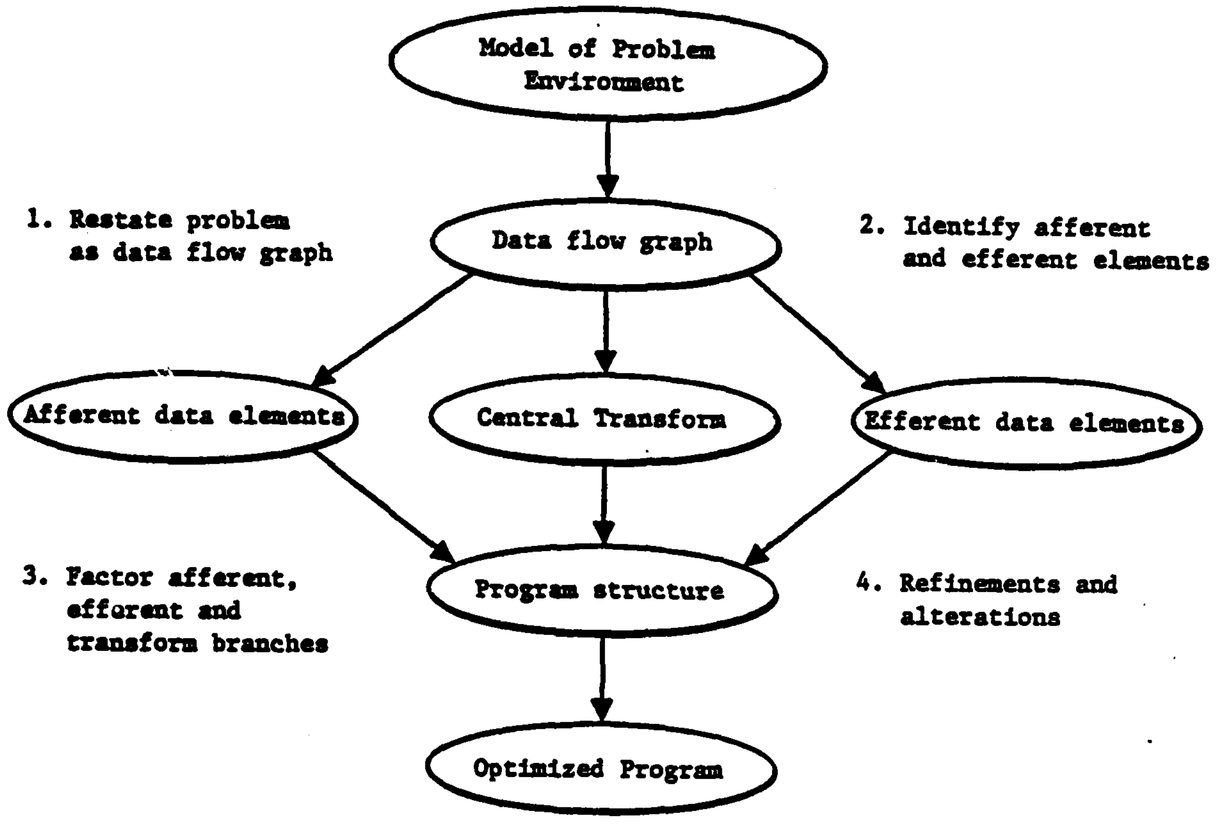

Model of Problem Environment
1. Restate problem as data flow graph
Data flow graph
2. Identify afferent and efferent elements
Afferent data elements
Central Transform
Efferent data elements
3. Factor afferent, efferent and transform branches
Program structure
4. Refinements and alterations
Optimized Program

Figure 3.2.4-1 Data flow design approach.

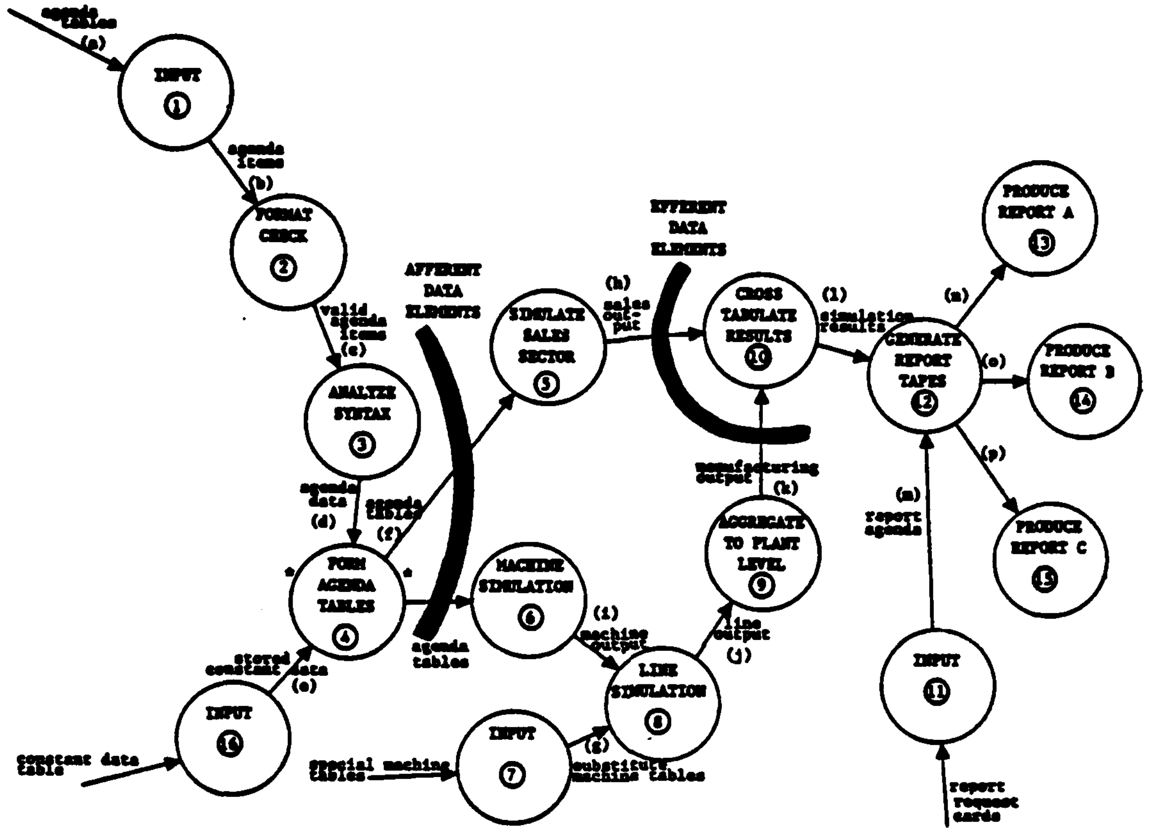


Figure 3.2.4-2 Data Flow Diagram for a Simulation System.

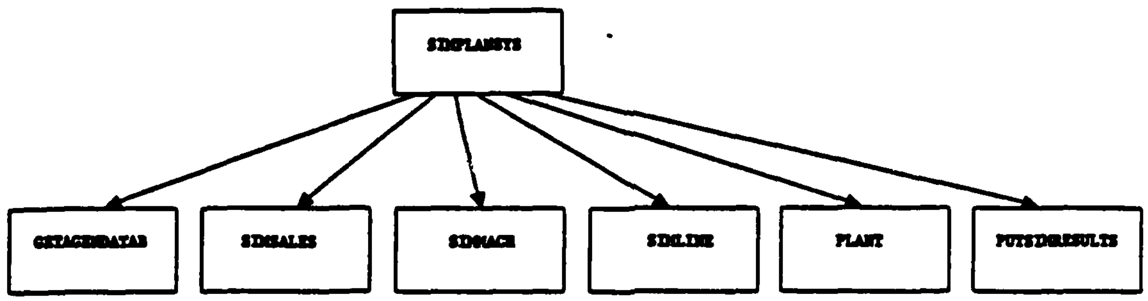


Figure 3.2.4-3 First-Cut Factoring.

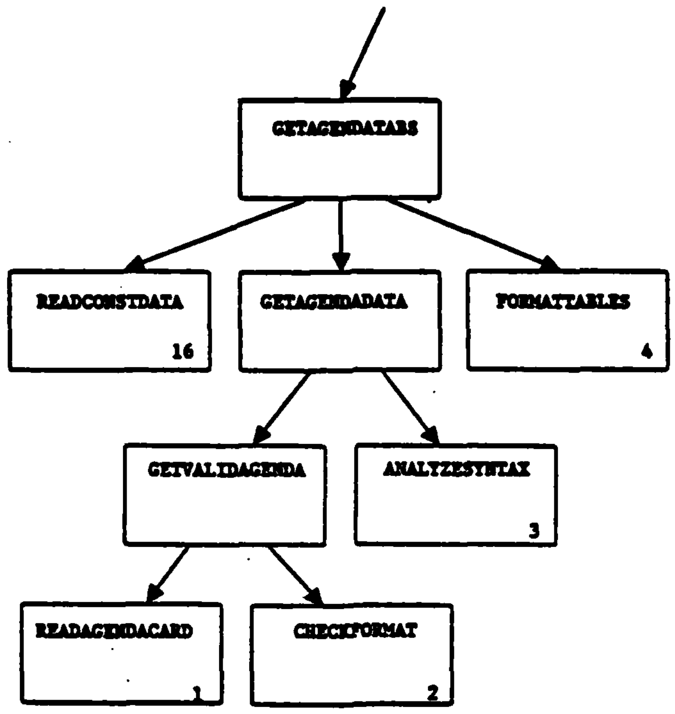


Figure 3.2.4-4 Factoring of an Afferent Branch.

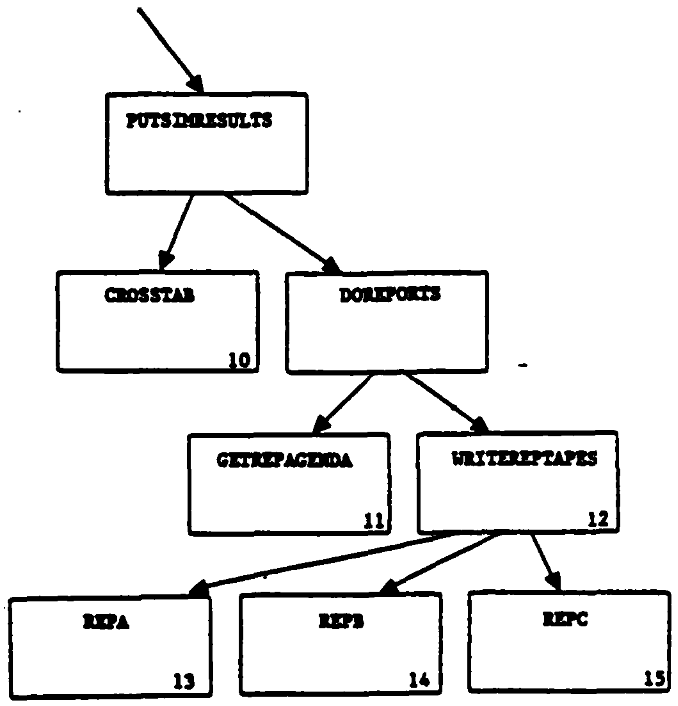


Figure 3.2.4-5 Factoring of an Efferent Branch.

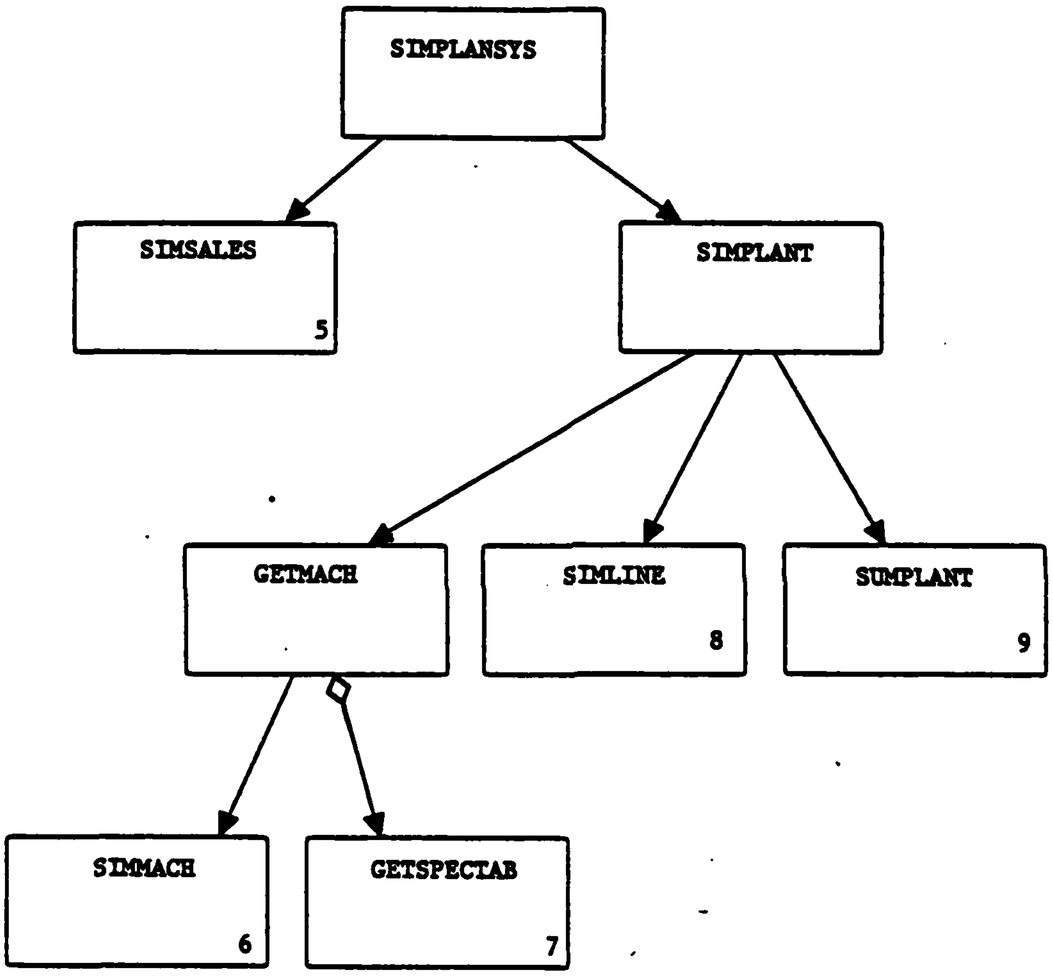


Figure 3.2.4-6 Trial Factoring of Central Transforms.

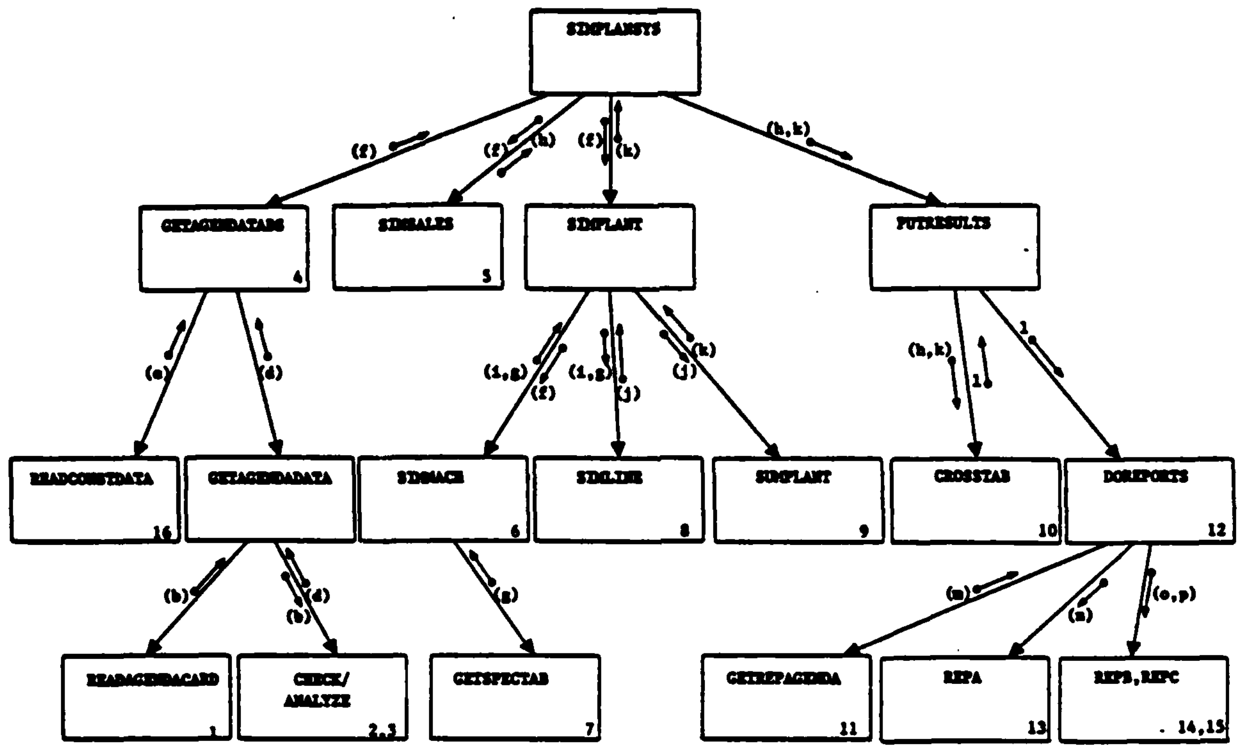


Figure 3.2.4-7 Possible Final Structure of Simulation System.

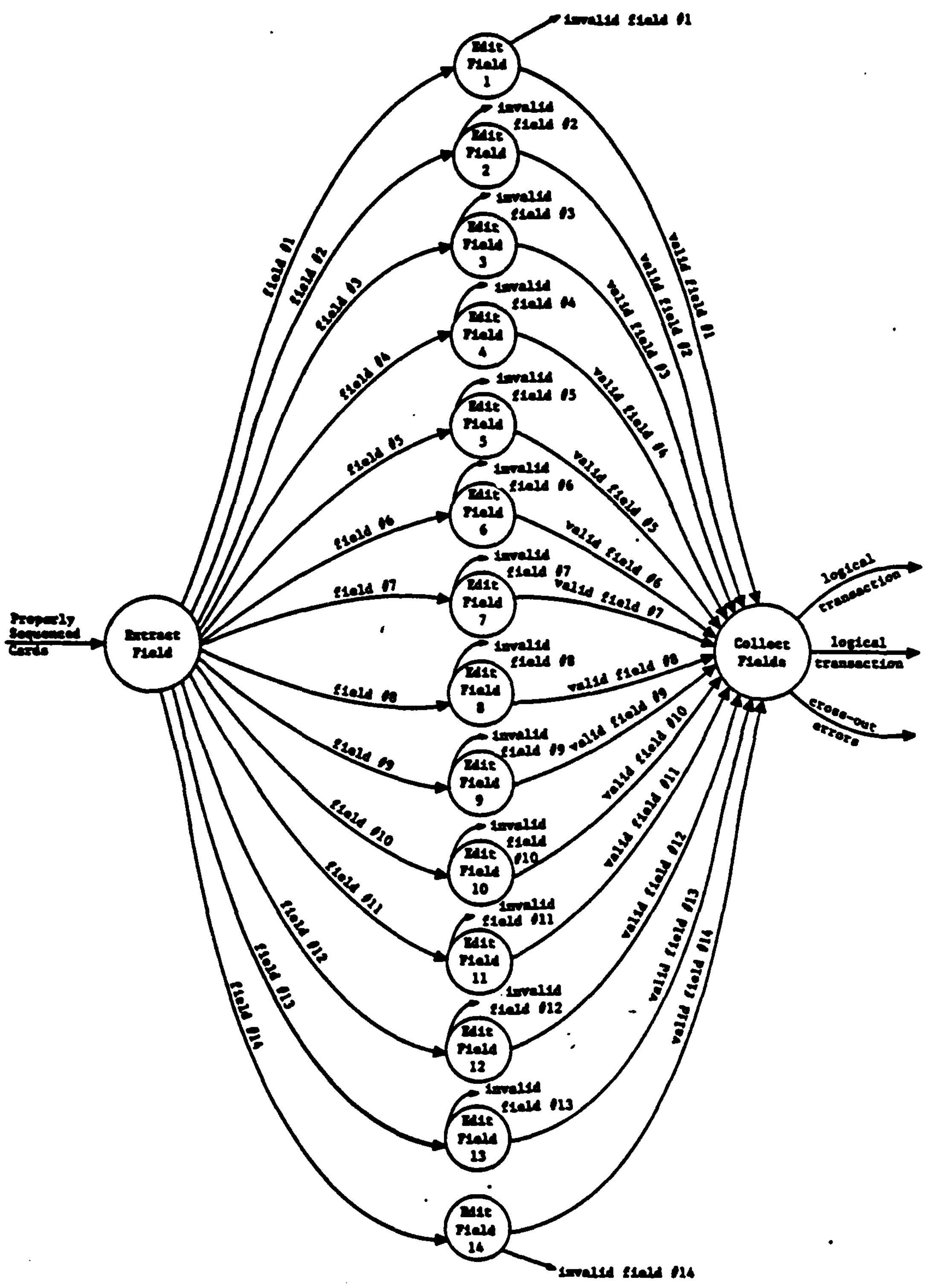


Figure 3.2.4-8 Expanded Data Flow Graph for Master File Update.

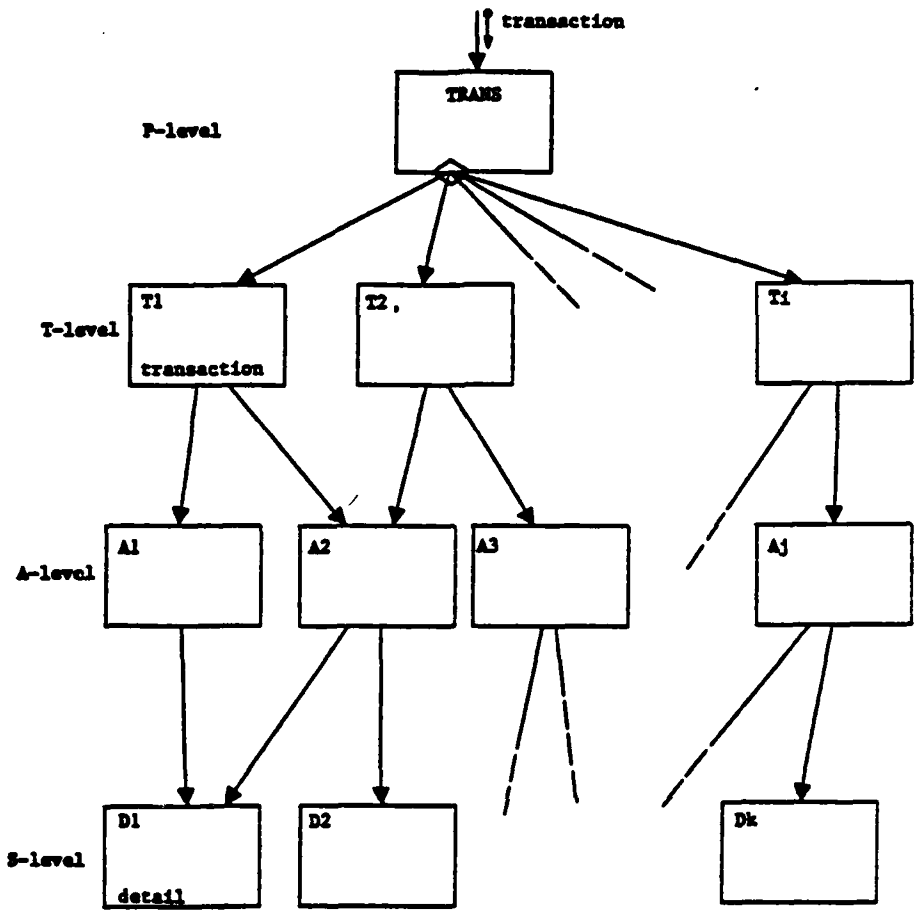


Figure 3.2.4-9 Model Structure Chart for Transaction-Centered System.

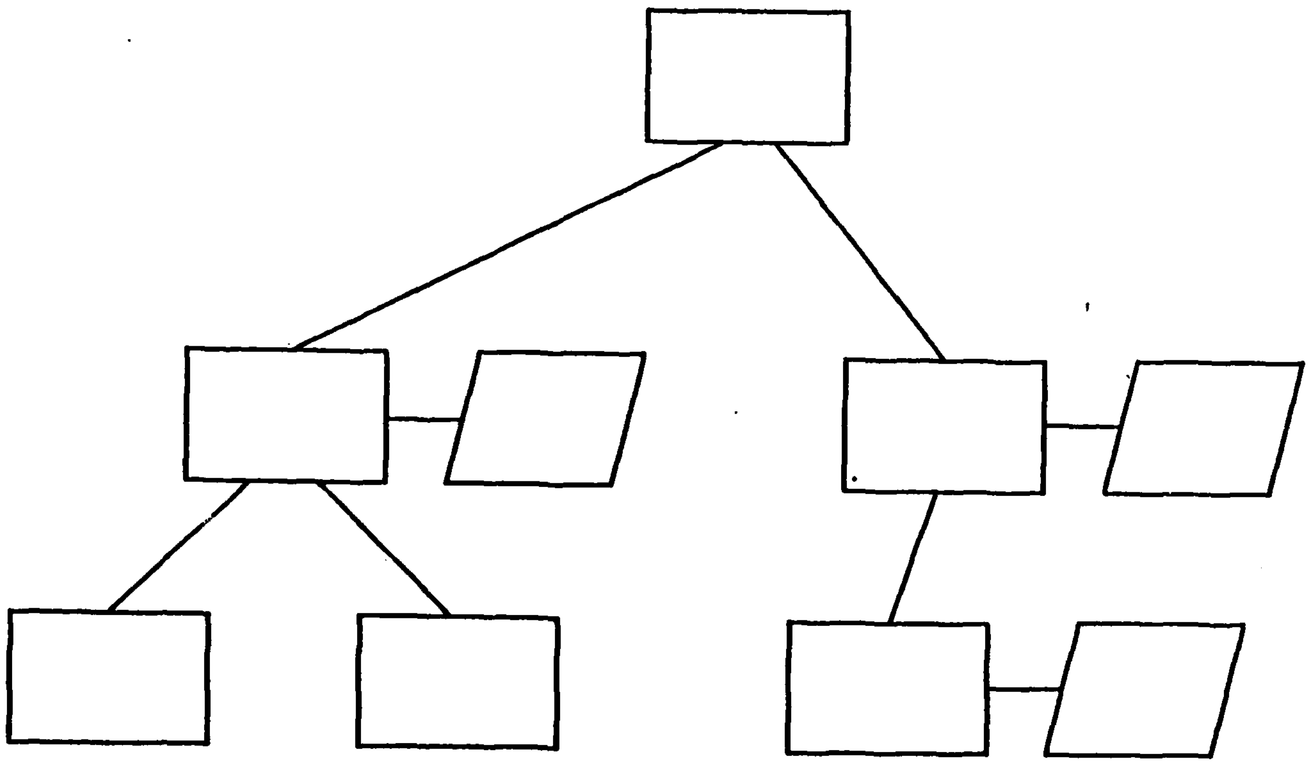


Figure 3.2.4-10 Example of a hierarchy chart incorporating parallel operations.

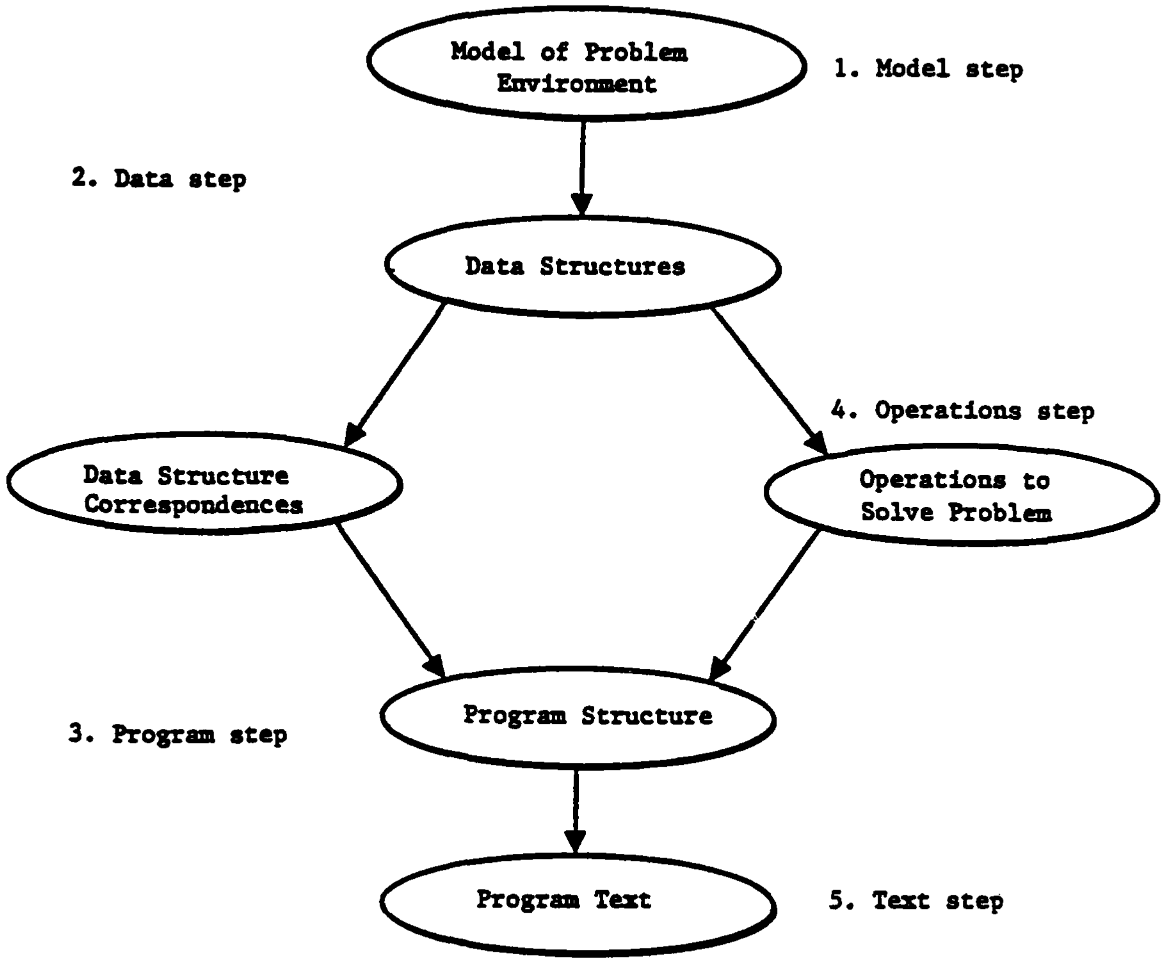


Figure 3.2.4-11 Jackson design approach.

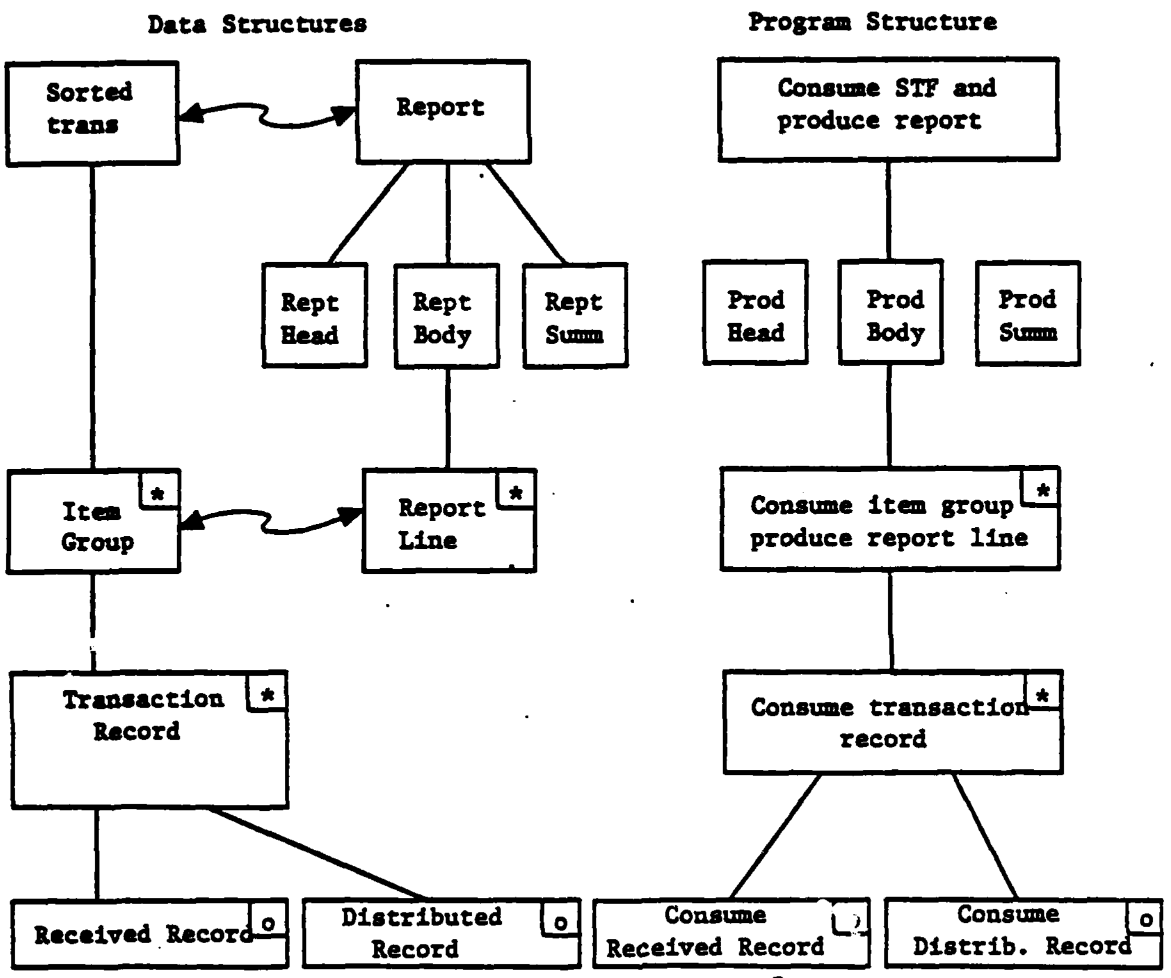


Figure 3.2.4-12 Data structure correspondences and program structure.

1. write heading
2. write net_chg
3. write item_grps
4. net_chg = 0
5. net_chg = net_chg + qty
6. net_chg = net_chg - qty
7. item_grps = 0
8. item_grps = item_grps + 1
9. open stf
10. read stf
11. close stf
12. critem = next item

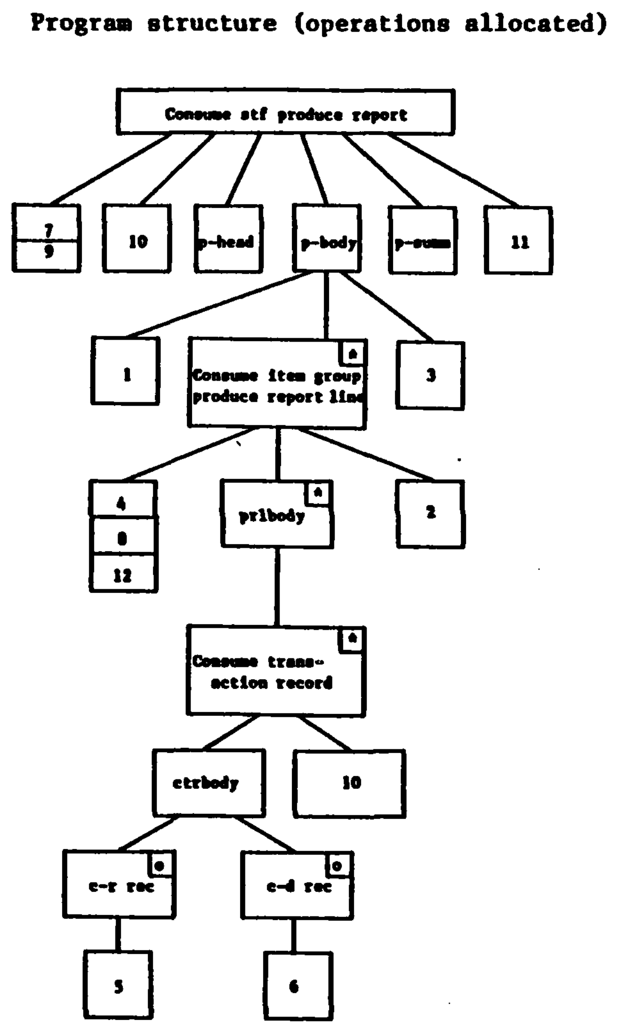

Program structure (operations allocated)
Consume stf produce report
7
9
10
p-head
p-body
p-summ
11
1
Consume item group produce report line
*
3
4
8
12
prlbody
*
2
Consume transaction record
*
ctrbody
10
c-r rec
o
c-d rec
o
5
6

Figure 3.2.4-13 Program structure with operations allocated.

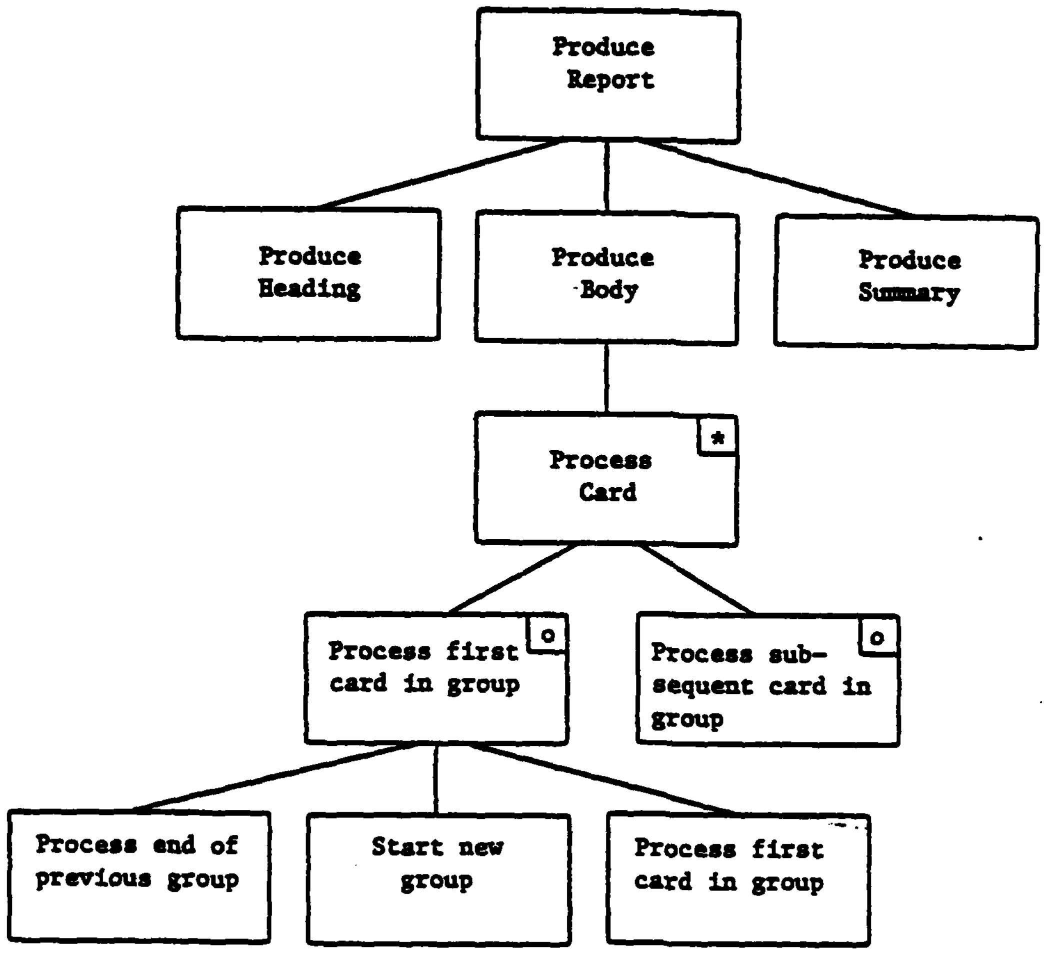

Produce Report
Produce Heading
Produce Body
Produce Summary
Process Card
*
Process first card in group
o
Process sub-sequent card in group
o
Process end of previous group
Start new group
Process first card in group

Figure 3.2.4-14 Functional Decomposition.

Step 5 - Identify and explain all departures from normal transform analysis that may be required because of the nature of the problem.

b. Transaction Analysis steps

Step 1 - Identify the sources of transactions in the DFD. Look for routing and/or distributed functions in the DFD. Figure 3.2.4-8 illustrates a DFD transaction center.

Step 2 - Specify the appropriate transaction-centered organization. Figure 3.2.4-9 is an example of how a transaction center might be organized.

Step 3 - Identify the transactions and their defining actions. The actions associated with transactions originating from the environment may be adequately defined in the Ada Requirements Document. However, the designer must define the actions associated with internally generated transactions not addressed in the Ada Requirements Document.

Step 4 - Note potential situations in which modules can be combined.

Step 5 - For each transaction, or cohesive collection of transactions, specify a transaction module to completely process it. Avoid excessive groupings of transactions into one transaction processor.

Step 6 - For each action in a transaction, specify an action module subordinate to the appropriate transaction module(s).

Step 7 - For each detailed step in an action module, specify an appropriate detail module subordinate to any action module that needs it.

Both transform analysis and transaction analysis are described in much greater detail in the Structured Design book authored by Yourdon and Constantine. Transform analysis is described in chapter 10, and transaction analysis is described in chapter 11. Examples of both strategies are provided in these chapters.

According to Yourdon and Constantine, transform analysis is preferred over transaction analysis. However, they do recognize that some systems are very much transaction-driven and suggest judicious application of transaction analysis. Perhaps a combination strategy which utilizes transform analysis as the primary driving force and utilizes transaction analysis as appropriate in the subproblems of the system is the optimum approach.

Guidelines for denoting concurrency on the system structure chart are as follows:

a. Tasks are represented by parallelograms (see Figure 3.2.4-10).

b. A dashed line is used for a directed arc from the activating module to the task. Whether the activation is intended to occur automatically or by an allocator should be noted in the documentation of the activating module.

c. A dashed line is used for a directed arc from a dependent task to the owning module. A notation should appear in the documentation of the activating module and in the owning module (they may often be the same module) as to the dependency of each task.

d. A solid line is used to connect modules and tasks to show synchronization and data transfer.

Figures 3.2.4-11, 3.2.4-12 and 3.2.4-13 illustrate a hierarchical program structured developed by utilizing the Jackson approach. Figure 3.2.4-14 shows the structure chart developed from a functional decomposition. Note that utilization of the Parnas approach or some new approach that may soon surface may give rise to a different hierarchical structure chart.

3.2.5 "Goodness" Criteria

3.2.5.1 Description

Intuitively, given two system designs that are both correct, one design may be better than the other because it is more easily maintained, it uses fewer resources, etc. In stating that one design is better than another, qualitative evidence may be presented. A set of metrics will be used as "goodness" criteria. The goodness criteria is important not only for evaluating designs; it can also be used to drive good design. Clearly, if a design analyst knows the criteria against which his/her product will be judged, he/she can make better design decisions initially.

Undoubtedly, there is not a universal set of design criteria that is equally applicable to all system designs. In fact, certain good design practices are often mutually exclusive and have to be traded off. While management should make the ultimate decisions regarding the goodness criteria, recent evidence evaluating the software life-cycle suggests that maintainability should be the last design objective to be sacrificed (except when real-time response constraints must be met). This philosophy is certainly consistent for the DoD in their embedded systems work. Consequently, a minimum set of metrics will be proposed to support the criteria for improved maintainability, but the set should in no way be considered complete in light of differing management objectives.

Two structured design metrics will be used to evaluate and drive each design step in an iterative cycle. The measures are cohesion and coupling (Yourdon and Constantine). The basic philosophy behind these measures is that a higher degree of maintainability is achieved as the modules that comprise the software design become more independent. Independence is measured by how well the elements comprising a module are related (cohesion) and to what degree the various modules comprising the system are interrelated (coupling). Typically, better independence is achieved when cohesion is maximized, and coupling is minimized. Descriptions of coupling and cohesion are provided below.

a. Cohesion - As stated above, cohesion is a qualitative measurement of the relationship among the elements within a single module. Using Myers terminology cohesion is stated in terms of module strength, with the strongest (most cohesive) module being of informational or functional strength, and weakest module (least cohesive) being of coincidental strength. The seven categories of cohesion, from highest to lowest strength, are:

7. Informational strength
6. Functional strength
5. Communicational strength

4. Procedural strength
3. Classical strength
2. Logical strength
1. Coincidental strength

The scale above is based on a study by Constantine in which he evaluated modules in specified classes with respect to program maintainability, extensibility, independence, error-proneness, and module reuseability. Consequently, the scale indicates a relationship among modules, inferring for example that procedural strength modules are "better" than classical strength modules. Note, however, that this scale is a guideline and peculiar aspects of a problem may dictate that a lower strength module is appropriate (the designer should, however, be able to defend a decision to use something other than an informational or functional strength module).

Presented below is a definition and example of module strength for each element on the scale. A more elaborate discussion is presented in Myers.

1. Coincidental-strength module: is a module whose function cannot be described (other than by the logic that comprises the module) or one that performs multiple, completely unrelated functions.

2. Logical-strength module: is one that performs a set of related functions, one of which is explicitly selected by the calling module (note: a logical strength module has a single interface for multiple functions).

3. Classical-strength module: is one that performs multiple sequential functions where there is a weak, but nonzero, relationship among all of the functions. Common examples are "initialization" and "termination" procedures.

4. Procedural-strength module: is a module that performs multiple sequential functions, where the sequential relationship among all the functions is implied by the problem or application statement. An example of a procedural-strength module would be one that, in response to an invalid transaction, would skip to the next transaction, checkpoint the master file, and display an error message on the operator console.

5. Communicational-strength module: is one that performs multiple sequential functions, where the sequential relationship among all the functions is implied by the problem or application statement <u>and</u> where there is a data relationship among all of the

functions. For example a module that, in response to an invalid transaction, "prints the transaction and copies it to the audit file" exhibits communicational strength.

6. Functional-strength module: is a module that performs a single, specific function.

7. Informational-strength module: is a grouping or package of functional strength modules for the purpose of information hiding.

b. Coupling - Coupling is a measure of strength of interconnection (i.e., communication bandwidth) between modules. Coupling is a measure that is both qualitative and quantitative. The number of parameters passed to a module is a quantitative measure of coupling. The type of interfaces between modules is a qualitative measure of coupling. Also, when coupling is minimized, implying the weakest strength interconnection model, the greatest module independence is achieved. The seven categories of module coupling from lowest (best) to highest (worst) strength are:

1. No direct coupling
2. Data coupling
3. Stamp coupling
4. Control coupling
5. External coupling
6. Common coupling
7. Content coupling

Again, the original classifications and ordering of coupling were based on a study by Constantine in which he evaluated modules in specified classes with respect to program maintainability, extensibility, independence, error-proneness, and module reuseability. Myer's coupling scale above is very similar to Constantine's original classifications and ordering of coupling. As was the case with the cohesion (module strength) scale, the coupling scale implies an order of desireability among the types of coupling and the design engineer should be prepared to justify his/her decision should something less than data coupled modules be designed.

Presented below is a definition and example of module coupling for each element on the scale. A more elaborate discussion is presented in Myers.

1. Content-coupling: two modules are content coupled if one directly references the insides of the other or if the normal linkage conventions between the modules are bypassed. For example, two assembly-language modules A and B are content coupled if A

references a word at some numerical offset from the beginning of B.

2. Common-coupling: occurs between a group of modules that reference a global data structure. The FORTRAN COMMON area exemplifies common coupling.

3. External-coupling: a group of modules are external coupled if they are not content or common coupled and if they reference a homogeneous global data item. External coupling occurs in FORTRAN with the use of labeled common areas.

4. Control-coupling: two modules are control coupled if they are not content, common, or external coupled and if one module explicitly controls the logic of the other, that is, one module passes an explicit element of control to the other module. Examples of "elements of control" are function codes transmitted to logical-strength modules, control-switch arguments, and module-name arguments.

5. Stamp-coupling: two modules are stamp coupled if they are not content, common, external, or control coupled and if they reference the same nonglobal data structure. Stamp coupling is similar to common coupling except the shared data structure is not global.

6. Data-coupling: two modules are data coupled if (1) they are not content, common, external, control, or stamp coupled, (2) if the modules directly communicate with one another (one directly calls the other), and (3) if all interface data between the modules are homogeneous data items.

7. No-direct-coupling: as the name suggests this is a module that does not have any intermodule relationships.

3.2.5.2 Guidelines

Once a structured design decomposition has been completed three steps should be taken to review the design before proceeding. These steps involve the application of the coupling and cohesion metrics to improve the design and to verify its quality ("goodness"). The steps are: post analysis of packaging, static design review, and dynamic design review (Myers).

a. Post Analysis of Packaging

With the object-oriented development available from the first part of the design effort, one analyzes the hierarchy chart to identify those objects that can be collected into packages. This reconciliation phase will result in the identification of objects utilizing a special notation, patterned after Booch, that will be folded back into the original structure or hierarchy chart.

Post analysis of object oriented design implies reviewing the design to ensure that module independence in the program has been maximized. The most important vehicle for doing this is the informational strength module or Ada package. The design should be reviewed looking for implicit or explicit assumptions and dependencies among modules, complex design decisions that can be isolated, or operations on common data that could be abstracted. These and other guidelines can be guided by the pre-packaging analysis described above. When these dependencies or complexities exist, or when suggested by the pre-packaging analysis, informational strength modules (Ada packages) should be constructed to eliminate dependencies and enhance the design. Parnas suggests the following guidelines in identifying data or functions to be collected:

1. Each module should hide some design decision from the rest of the system.

2. A data structure, its internal linkings, accessing procedures, and modifying procedures should comprise a single module.

3. The format of control blocks (data structures) must be hidden.

4. Character codes, alphabetic orderings, and similar data should be hidden.

5. The sequence in which certain items will be processed should (as far as practical) be hidden within a module.

6. Preferred reason for packaging is to hide some aspect of the implementation within the package-body; this may be data structures, data types, common subprograms, or common packages.

7. A good reason for packaging is in support of abstract data types in which the type must be made visible in the package-declaration but its physical structure may be hidden in the private portion.

8. Another less compelling but valid reason for packaging is to gather together objects which are referenced by the same set of other modules, to reduce the number of library units.

Data or functions which are collected are graphically depicted as shown in Figure 3.2.5-1. Windows represent entry points into the objects realized by the graphical notation. The graphical object is given a name and the entry points are also appropriately named. This description of the objects can then be realized in the hierarchy chart as shown in Figure 3.2.5-2. Note, that the interconnectivity is still intact from the original chart or if data flow is represented on the arcs, this flow is still intact.

b. Static Design Review

Static design review is simply an inspection of the completed design with respect to a set of criteria and questions. The analysis is best carried out by an independent reviewer with the intent of uncovering design weaknesses, not to make immediate improvements. The correction of uncovered weaknesses is best delayed until after the review is complete so that hasty or ill-conceived changes are not incorporated into the design.

Myers presents the following set of questions as being reasonable for a static review:

1. Does each module have functional or informational strength? If any do not, then why not?

2. Is each pair of modules either data coupled or not directly coupled? If any are not, why not?

3. Is the decomposition complete? That is, can the logic of each module be easily visualized?

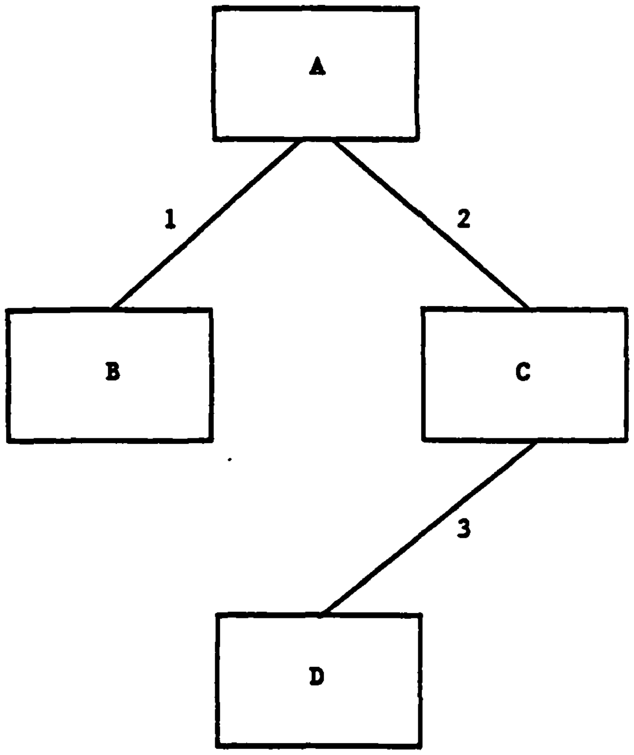
4. When a module or entry point has a fan-in greater than 2 (i.e., multiple immediate-superordinate modules), are the interface definitions to the module consistent? That is, does each interface to the module have the same number of input arguments and the same number of output arguments? Does each corresponding argument in each interface have the same attributes?

5. Is there any unnecessary redundancy in any interface?

6. Is the interface data to each module consistent with the definition of the module's function?

7. Are there multiple modules in the system that appear to have the same function?

8. Is each module described by its function rather than by its context or logic? Has each function been stated accurately?



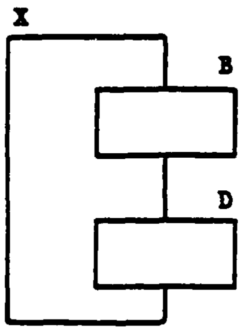


Figure 3.2.5-1 Notation for packaging with windows representing package operations.

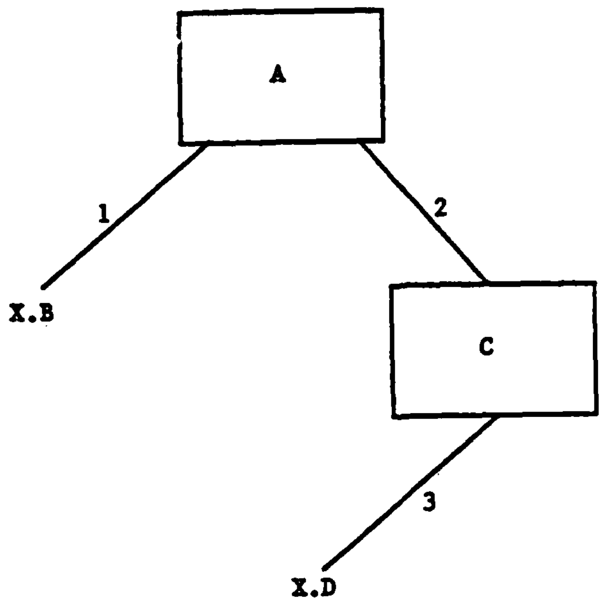


Figure 3.2.5-2 Incorporation of packaging notation into hierarchical representation.

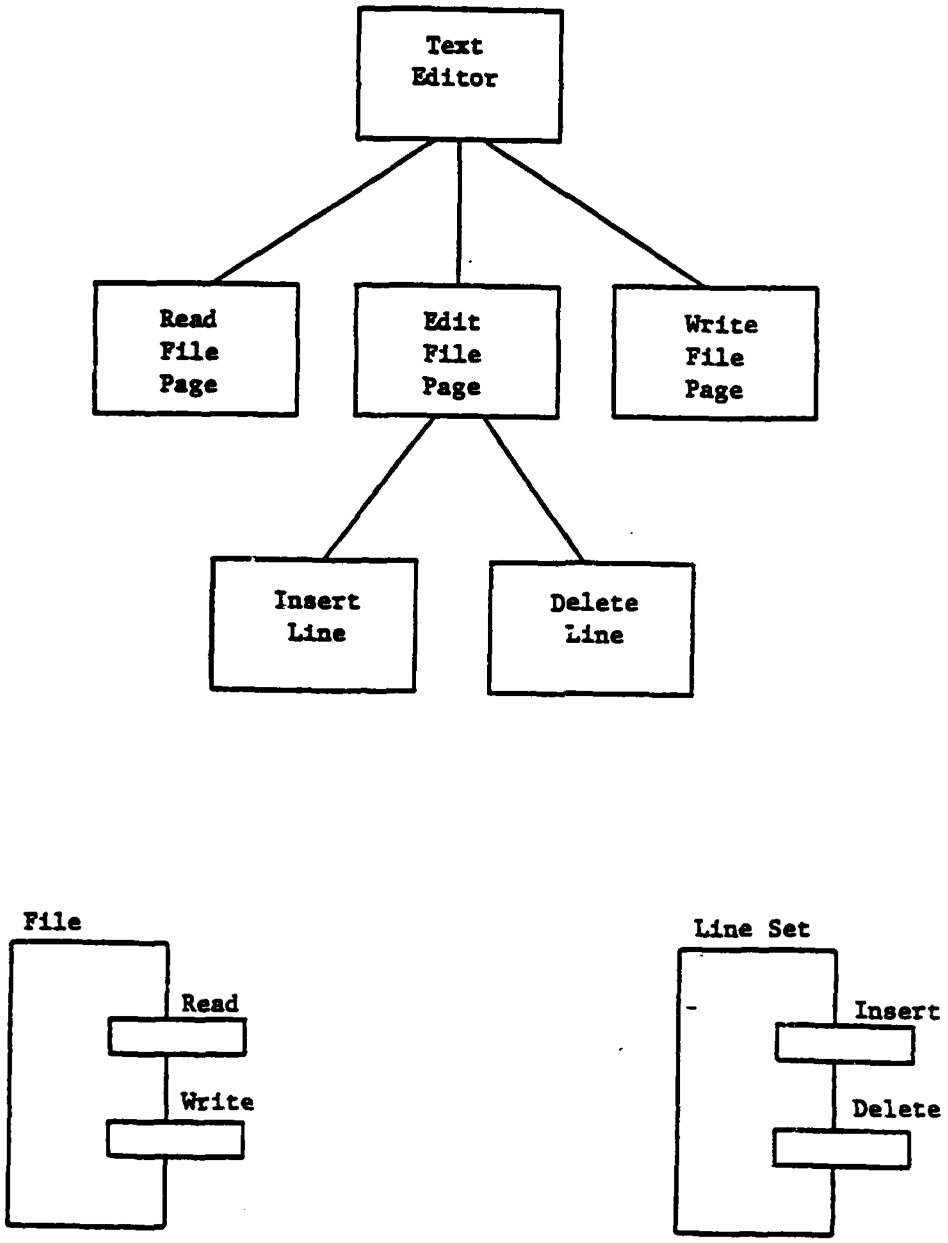


Figure 3.2.5-3 Collection of packages and notational aspects.

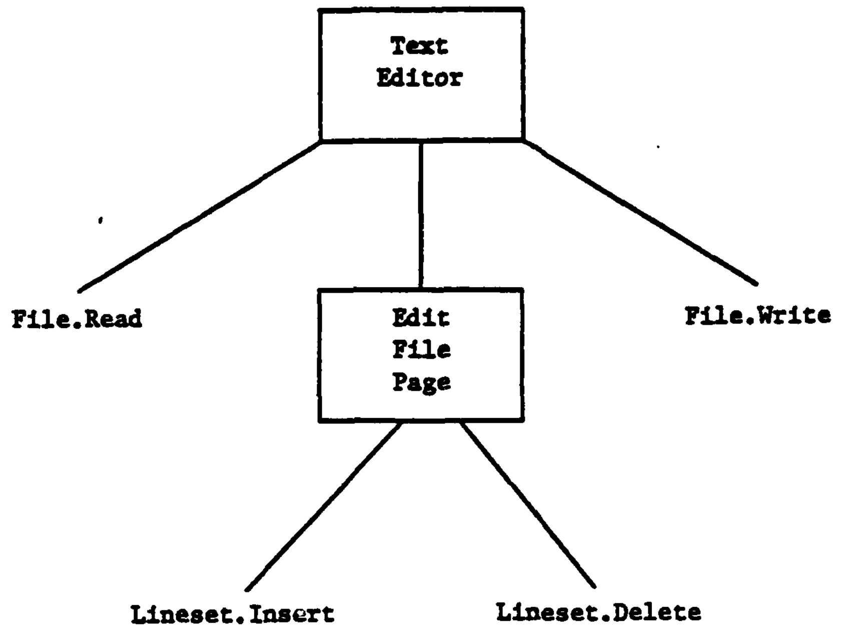


Figure 3.2.5-4 Incorporation of packaging into structure chart.

9. Are there any pre-existing modules that could have been used in this system?

10. Are there any restrictive modules (i.e., modules that are unnecessarily overspecialized)?

11. Are all modules predictable (i.e., do the same inputs produce the same outputs for each invocation)?

12. Is the design realizable by the Ada programming language?

13. Is the hierarchical structure too extreme (e.g., overly "short and fat" or "tall and lean")? Note, such extreme structures do not necessarily imply problems, but they should be explored because they could indicate that the decomposition used was incorrect.

14. Is the design correct (i.e., are all items specified in the requirements document readily identified in the system design)?

15. Have rules of parsimony been followed? That is, does the system do only that stated in the requirements document?

16. Is the design obscure? Does it contain anything that might easily be misunderstood?

17. Have any unstated assumptions been made?

c. Dynamic Design Review

This is a mental "walk-through" of the program. It begins by developing a set of tests and then tracing the state of the system as it would change in response to the steps of the test. Since some logic might not exist at the time of the dynamic review, the functionality of "stub" modules can be assumed to be correct. The purpose of the review is to find design flaws such as: missing or incomplete functions; erroneous interfaces; incorrect results; etc.

See Figures 3.2.5-3 and 3.2.5-4 for examples of the application of some aspects of the above discussion pertaining to the reconciliation. Clearly, the application of coupling and cohesion is subjective at best but by utilizing the above considerations one should determine a coupling and cohesion level for each module and module relationship.

3.2.6 Requirements-to-Design Traceability

3.2.6.1 Description

Traceability of requirements to design is a link in the chain of traceability from user specifications to actual implementation program units. The purpose of requirements-to-design traceability during development is two-fold. First, the tracing of requirements to design facilitates the evaluation of changes to the requirements during development of the system. Secondly, it provides some form of verification that the design does satisfy the requirements.

The traceability step occurs at two points in the design process. First, traceability is applied immediately after evaluating the Goodness of design (3.2.5). If it is determined at this point that the design does satisfy the requirements, the designer should proceed with step 3.2.1 (Data Structures).

The second time traceability is applied is during detail design at step 3.3.2 (Final Design Review). Of course, if any requirements cannot be traced to design at this point, the design must be corrected before completing the final design review and proceeding to the implementation phase.

3.2.6.2 Guidelines

Guidelines a through f are to be performed as tasks in the order specified. Whereas guidelines g and h are only suggestions.

a. Develop a matrix to map Ada Requirements Document components into Ada Design Document components.

b. Develop a set of Ada Requirements Document components to be mapped into Ada Design Document components. A possible set of Ada Requirements Document components is provided below:

   DIAGRAM BOX NUMBER
   NON-FUNCTIONAL REQUIREMENT
   CONCURRENCY REQUIREMENT

   It will be necessary to use some kind of identification mechanism to abbreviate actual non-functional and concurrency requirements on the traceability matrix.

c. Develop a set of Ada Design Document components to be used in the mapping process from requirements to design. A possible set of Ada Design Document components is provided below:

   SUBPROGRAM NAME
   PACKAGE NAME
   PACKAGE NAME.SUBPROGRAM NAME
   PACKAGE NAME.TASK NAME

d. Map each Ada Requirements Document component to an Ada Design Document component.

e. Do a reverse mapping from design to requirements after the initial mapping of requirements to design.

f. Use the results of the forward and backward mapping of requirements and design to verify that the design does satisfy the requirements.

g. Try to map each Ada Requirements Document component into as small an Ada Design Document component as possible.

h. Arrange the Ada Requirements Document components in categorical order on the traceability matrix. For example, map all the low-level functional blocks associated with each major function first (in major function order) and then map the non-functional and concurrency requirements in that order.

Table 3.2.6-1 is an example of a traceability matrix that illustrates the mapping of requirements to design.

Table 3.2.6-1 REQUIREMENTS-TO-DESIGN TRACEABILITY MATRIX

ADA REQUIREMENTS DOCUMENT COMPONENTS	ADA DESIGN DOCUMENT COMPONENTS
A111	INIT_SYSTEM
A112	REIN_MESSAGE
.	
.	
.	
A2NN	OP_INTERFACE
.	
.	
.	
A3NN	ASSM_MSG_PKG0.TASK1
.	
.	
.	
ANNN	MSG_SWITCH_PKG1.PROC1
NFRQMT #1	MSG_SWITCH_PKG2
	MSG_SWITCH_PKG3
CONCURR_RQMT #1	MSG_SWITCH_PKG4
	MSG_SWITCH_PKG5

3.2.7 Module Interconnectivity

3.2.7.1 Description

Module interconnectivity is described via an N^2 chart. The N^2 chart is constructed by utilizing a matrix as described in "The N^2 Chart", by R. J. Lano. It is a square matrix, with all functions on the diagonal, all outputs of a function are horizontal (left or right), all inputs are vertical (up or down), thus all non-diagonal (non-function) positions define one-way interfaces between the respective functions.

N^2 charts and interconnectivity charts are used to document the interfaces of the final architectural design. The structure chart is the mechanism that is used during multiple iterations of architectural design to evaluate and change the structure of the system.

3.2.7.2 Guidelines

a. Using the final structure chart of architectural design and beginning with the number 1, sequentially number all modules on the chart until all modules have been assigned a unique number.

b. Construct an N-squared chart, i.e., represent the modules along the diagonal in numerical order.

c. On the N-squared chart indicate which interfaces exist.

d. Adopt a numbering scheme for interfaces; possibilities include assigning unique sequential numbers left-to-right, top-to-bottom, or using matrix position outputting function number-dash-inputting function number.

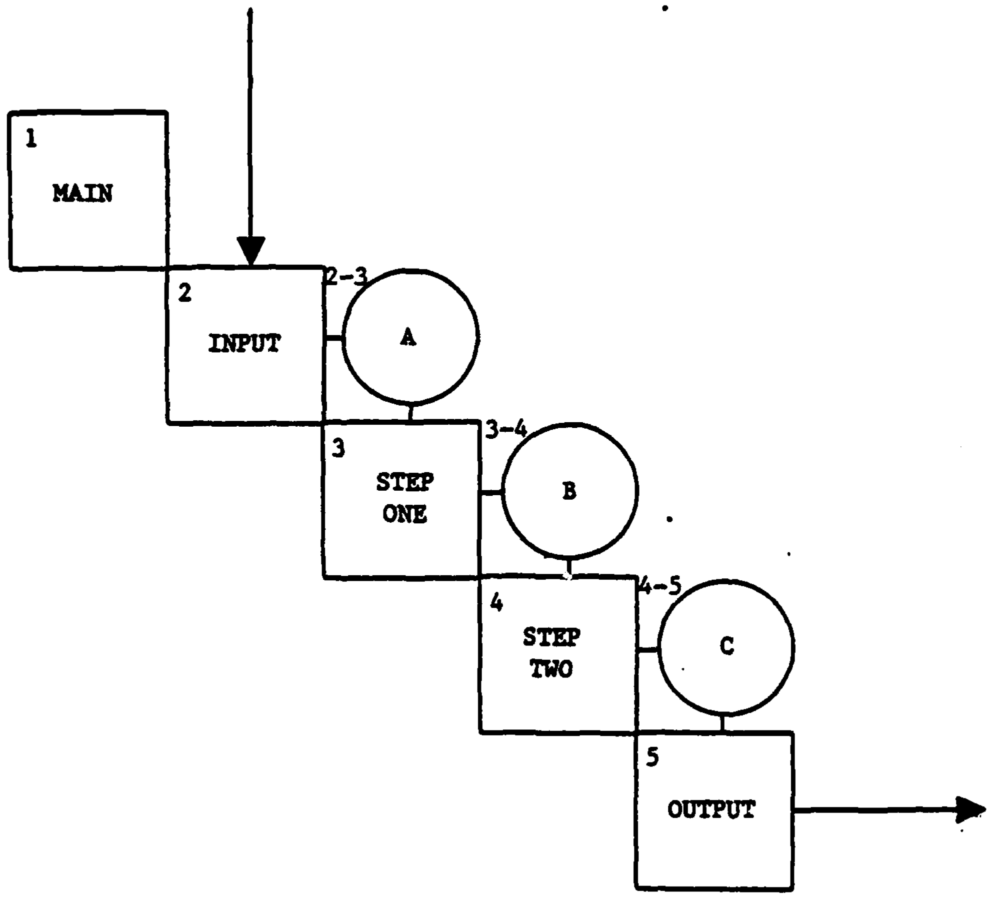


Figure 3.2.7-1 Example Sequence of Processes with Direct Data Links.

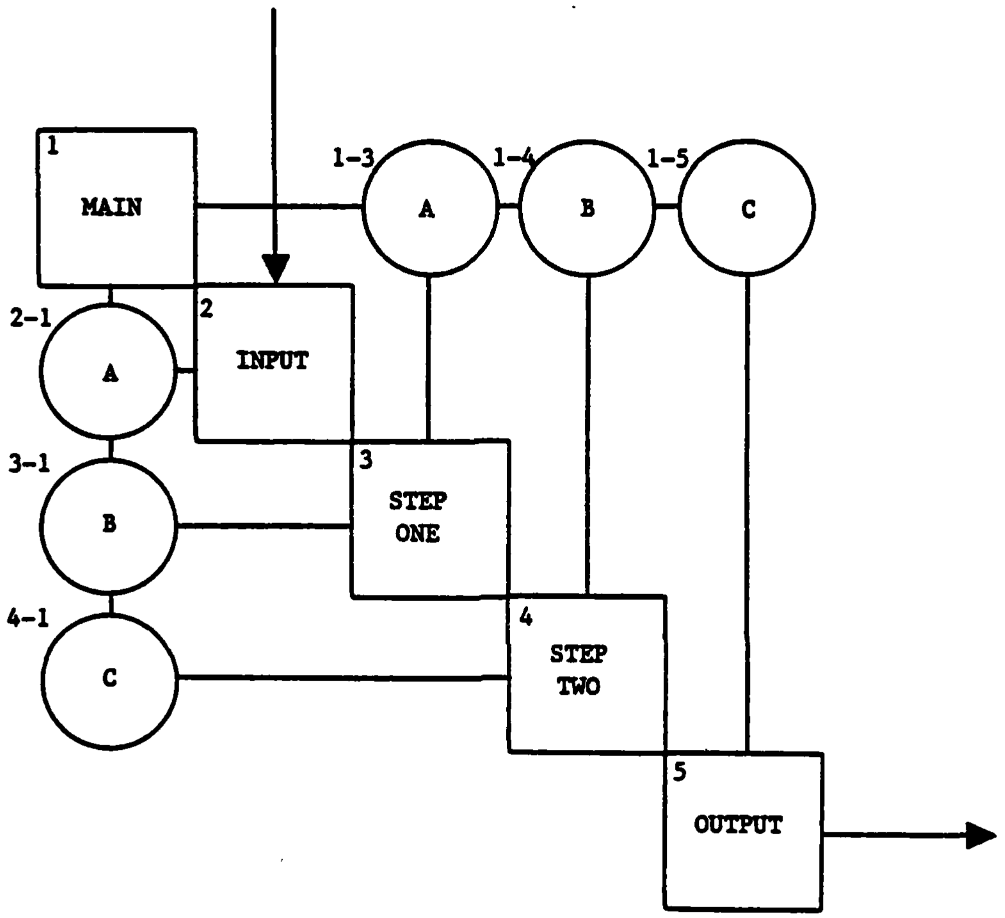


Figure 3.2.7-2 Example Sequence of Processes with Main Module Controlling all Data.

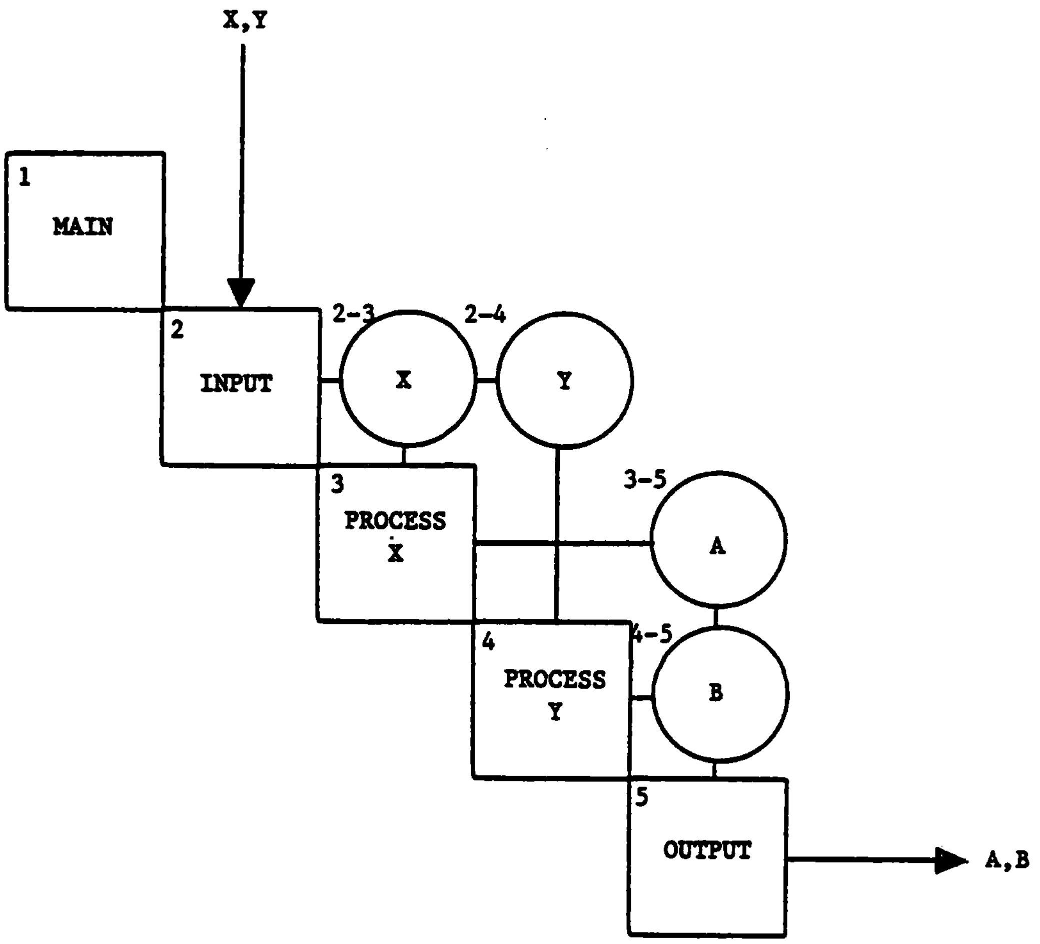


Figure 3.2.7-3 Example Transaction Processes.

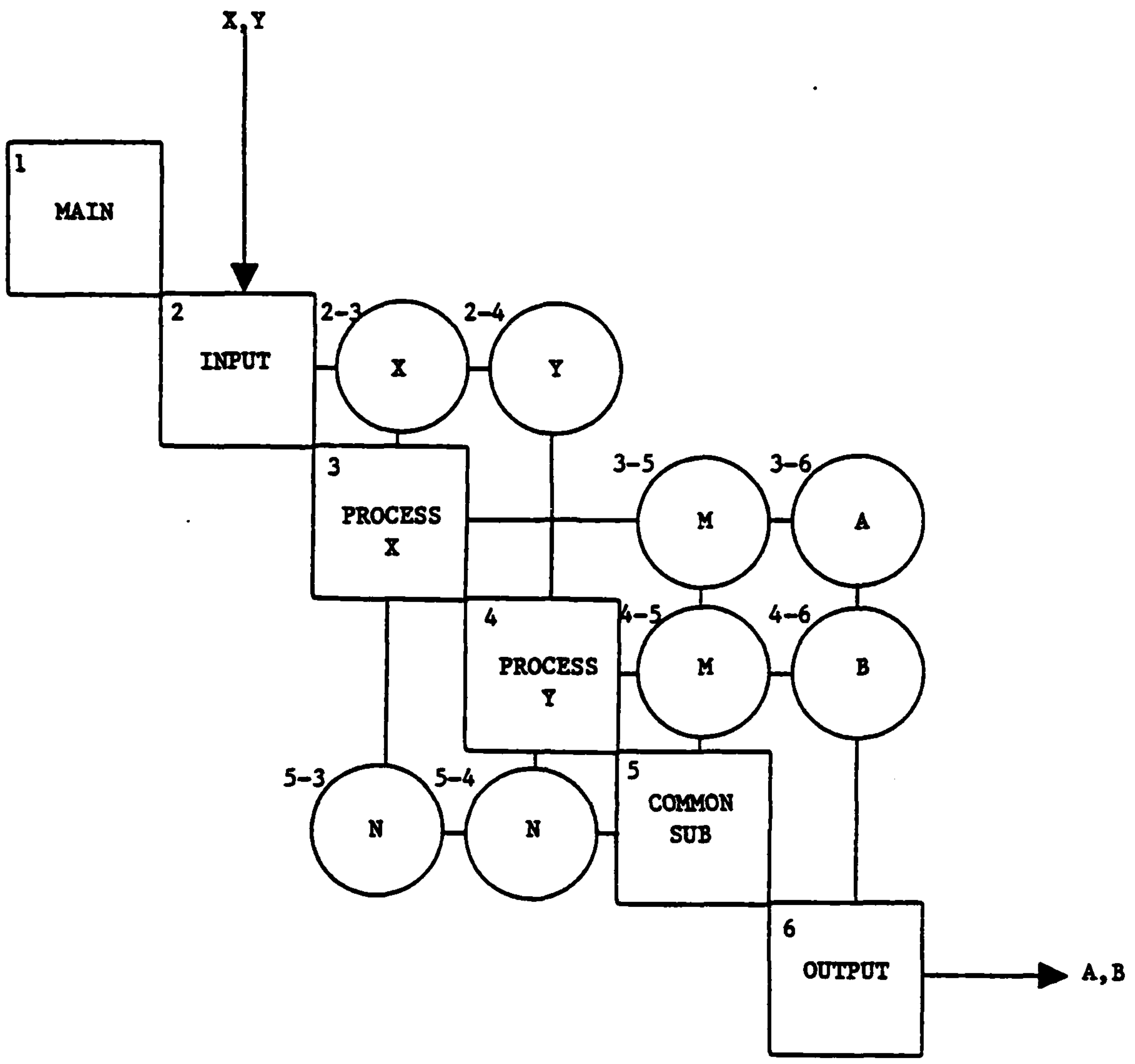


Figure 3.2.7-4 Example Processes Using a Common Subprocess.

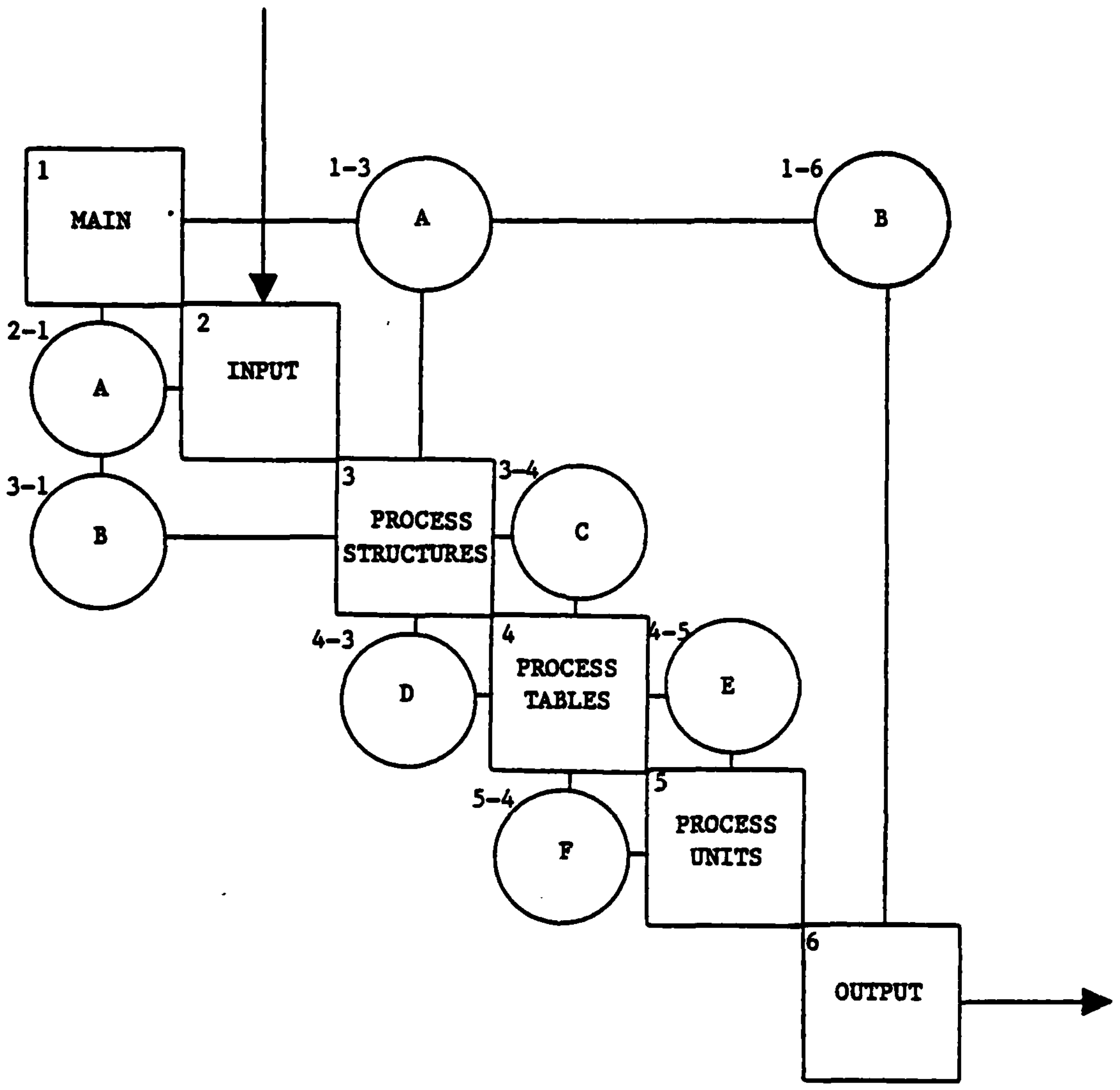


Figure 3.2.7-5 Example Multilevel Subprocesses.

3.2.8 Preliminary Design Review

The preliminary design review is an intermediate point in the design process where the designer stops and reviews the overall functional design developed to date. Any discrepancies noted at this point should be corrected through iterations of the appropriate design steps previously stated (3.2.1 - 3.2.7).

3.2.9 Ada Unit Specifications

3.2.9.1 Description

System architectural design as so far described, expressed and documented as structure charts, interconnectivity charts, N^2 charts, and package charts, can now be expressed using Ada constructs and program organization. Since AIM provides system design specification and development in terms of units and constructs compatible with Ada language design philosophy, system expression in Ada should be isomorphic to architectural design. The structure created during architectural design can be implemented with the Ada compilation units. The uppermost control module must be a subprogram as the Ada Reference Manual in Chapter 10 states only "a subprogram can be a main program". The remainder of the modules determined during architectural design may be represented by declarations in the packages selected during architectural design, or by subprogram declarations, or by body stubs in the parent modules if the implementations selected are subunits.

3.2.9.2 Guidelines

The guidelines for formulating Ada specifications from architectural design are listed below:

a. The main program or uppermost control module must be formulated as an Ada subprogram.

b. When formulating package declarations only include objects which must be visible to external modules.

c. When type declarations appear in package declarations, use the private portion of the package to protect the structure of the type from direct manipulation by external modules when possible.

d. Hide as many internal objects of packages within their respective bodies as possible so that they are hidden from external modules and do not influence the compilation of the package declaration.

e. A subprogram or package which is only referenced by one module may be nested within that one module and implemented as a subunit, or it may be declared as a library unit; as a subunit it would inherit the context of its parent, and as a library unit it would be accessible from other units through their context specifications.

f. Activation of a task, which can be controlled by point of declaration or by point of creation by an allocator, should be carefully selected.

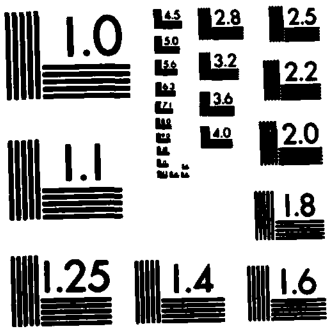
AD-A123 305 LARGE SCALE SOFTWARE SYSTEM DESIGN OF THE AN/TYC-39 STORE AND FORWARD MES..(U) GENERAL DYNAMICS FORT WORTH TX DATA SYSTEMS DIV 09 NOV 82 DAAK80-81-C-0108 2/2

UNCLASSIFIED F/G 9/2 NL

END

FILMED

DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

g. As detailed in Section 9.4 of the Ada Reference Manual a block or body with dependent tasks is not left until all dependent tasks have terminated; therefore, care must be exercised in assigning these dependencies which are controlled by the placement of task object declarations and access type declarations.

The following shall be the general format of the Ada program unit specifications:

a. Context declarations, as necessary

b. Unit specifications, including parameter lists

c. Prologue (as comments): Author, date, revision level; functional description and requirements traceability

d. Variable and aggregate type, range, and enumeration declarations

e. Subunit declarations, including parameter lists

f. Entry declarations for tasks

In addition, the following body declarations can be made as required:

a. Stubs for hidden procedures and functions

b. Definitions for hidden data.

3.2.9.3 <u>Example</u>

Example:

```
    with MESSAGE

    package MAILBOX is

    -- J. K. Writer
    -- 10 DEC 1981   REV A
    -- Ref. A231, A232

         type LINE-ID is private;

         procedure GET-LINE (RI:  in MESSAGE.ROUTE;
                             ID:  out LINE-ID);
         procedure RECEIVE (ID:   in LINE-ID;
                            MSG:  out MESSAGE.BUFFER);
         procedure SEND (ID:   in LINE-ID;
                         MSG:  in MESSAGE.BUFFER);

    private

         type LINE-ID is new INTEGER;

    end MAILBOX;
```

3.2.10 Hardware/Software Partitioning

3.2.10.1 Description

3.2.10.1.1 Rationale. AIM is intended to be applied to the design of complex embedded systems, typically consisting of fixed, pre-defined hardware components and functionally integrated by programs operating on one or more digital processors. At some point in the process of system definition, system functions will be distributed among hardware and software components. There is some degree of variability with respect to when in the system definition process such allocation takes place. One reason for this is that performance evaluation standards are still evolving: a rigorous performance evaluation methodology will enable designers to rationally distribute hardware and software functions based on performance, cost, resource, and procurement constraints. Project management generally desires hardware allocation to be made as early as possible so that procurements can be initiated in a timely manner.

3.2.10.1.2 Preliminaries to Hardware/Software Partitioning. This section addresses the design status that should exist at the time of allocation of system functions to hardware and software. Hardware/software partitioning (HSP) is the last task of architectural design prior to detail design. This concept of HSP is compatible with general AIM design philosophy. The object-oriented architectural design allows early identification of performance-critical attributes. The AIM methodology provides for software prototyping of critical performance functions prior to any detail design.

The following is a list of inputs, or design states, to HSP activity:

a. Major functional partitioning of the system - structure chart of major functional modules and N^2 charts showing module interfaces.

b. High-level concurrency model showing task structure.

c. Performance requirements including:

   1. Volume of data transmission (internally and to external hardware)

   2. Processing constraints in terms of data transformations within system-defined event intervals

   3. Identification of time-critical functions

   4. I/O requirements including resource requirements, access frequency, media requirements

5. Identification of likely modifiable functions (including adaptability requirements given by system spec)

6. Identification of hardware control functions (including display and operator communication) for possible assignment to interface hardware

7. Software allocation - those functions definitely assigned to software, including major utility and library functions

8. Determination of redundancy and diagnostic requirements.

d. Data structures.

1. Model of logical data structures (developed during requirements and step 2 of architectural design).

2. Jackson data structures.

3.2.10.2 Guidelines

3.2.10.2.1 Factors. Factors that influence the allocation of system functions to hardware and software are considerations of flexibility, efficiency, cost, resources, level of effort, etc. More specifically, design questions pertain to the following areas:

a. Performance and resource constraints. As a result of requirements analysis, the implications for performance evaluation of processing requirements will be specified. Processing requirements include internal storage capacity, data transmission capacity (in terms of rate and volume), file capacity, general features of file organization and associated access methods, display requirements, operator interface response time, and concurrency required by system definition. System design reflects solutions to the problems imposed by these constraints; distribution of hardware and software functions is an expression of these solutions.

b. Maintainability. The analysis of a complex embedded system with an extended life cycle must consider questions such as the following: What system functions are most specialized, less general? What design features will be most sensitive to requirements specification changes? Which functions are generalizable from current design to a broader set of applications in the same class? (More specifically, for this contract: To what extent have general characteristics of C^3 systems been taken into account?)

The implication of such questions for hardware/software partitioning is that the more generalized a system function is, or the more likely to require modification, the more likely it is to be assigned to software, other things being equal.

c. Adaptability. The application environment of the AN/TYC-39 message switch consists of various types of external hardware to which the system must interface. The system must be adaptable to changes in system configuration. The definition of system status includes both the maintenance of information necessary for recovery/restart and for the definition of system configuration. HSP decisions must take account of the associated factors of reliability, security, and reconfigurability.

3.2.10.2.2 Types of Hardware Allocation in the System. The purpose of this section is to informally classify the broad types of hardware functions in a total system.

a. Hardware fixed by system specs. This is external hardware to which the embedded computer system must interface. Characteristics of the external hardware are reflected in the processing system specs. Design options here involve the specification of mechanisms at intermediate interface levels, between the fixed hardware and the processing system.

b. System architecture. The architecture of the embedded computer system (the organization or configuration of its hardware components) is a design feature specified as (1) a solution to performance constraints and (2) a reflection of "good" design criteria. Part of the specification expresses the choice between uni- and multi-processor architecture. Multiprocessor configuration can be a solution to timing constraints and concurrency requirements, and can physically embody high-level design of tasking architecture.

c. Special purpose hardware functions. System functional organization identifies groups of functions which can be allocated to either hardware or software. Choice of allocation, as indicated above, would ideally be made on the basis of a methodical performance analysis. In the absence of a well-defined methodology, hardware/software allocation must be based on available information on performance requirements and constraints, level of effort, and system adaptability factors (reconfiguration requirements). Availability of off-the-shelf components for given system functions will influence allocation decisions.

3.2.10.2.3 HSP Subtasks. This section addresses the specific tasks involved in the hardware/software partitioning activity.

a. A major task is the allocation of software functions to control processing hardware. Choice of uni- and multi-processor architecture will determine features of detail design, as well as the nature of supervisory functions of system software.

b. Input/output control and processing functions must be assigned to device drivers.

c. Memory access modes satisfying data access performance requirements must be determined. Decisions regarding data transmission and data base access hardware must be made and the corresponding software performance requirement stated.

d. Redundancy and diagnostics provisions must be allocated.

e. System representation shall include graphical or schematic display of the following:

    1. Physical arrangement or configuration of internal hardware components, including processing units, program storage units, data base storage units, data and signal transmission and control units, data and signal paths, I/O control units, display units, and operator devices.

    2. A system structure chart as described in 3.2.4, with an indication of functional assignment to hardware, as by drawing a double line around modules or groups of modules.

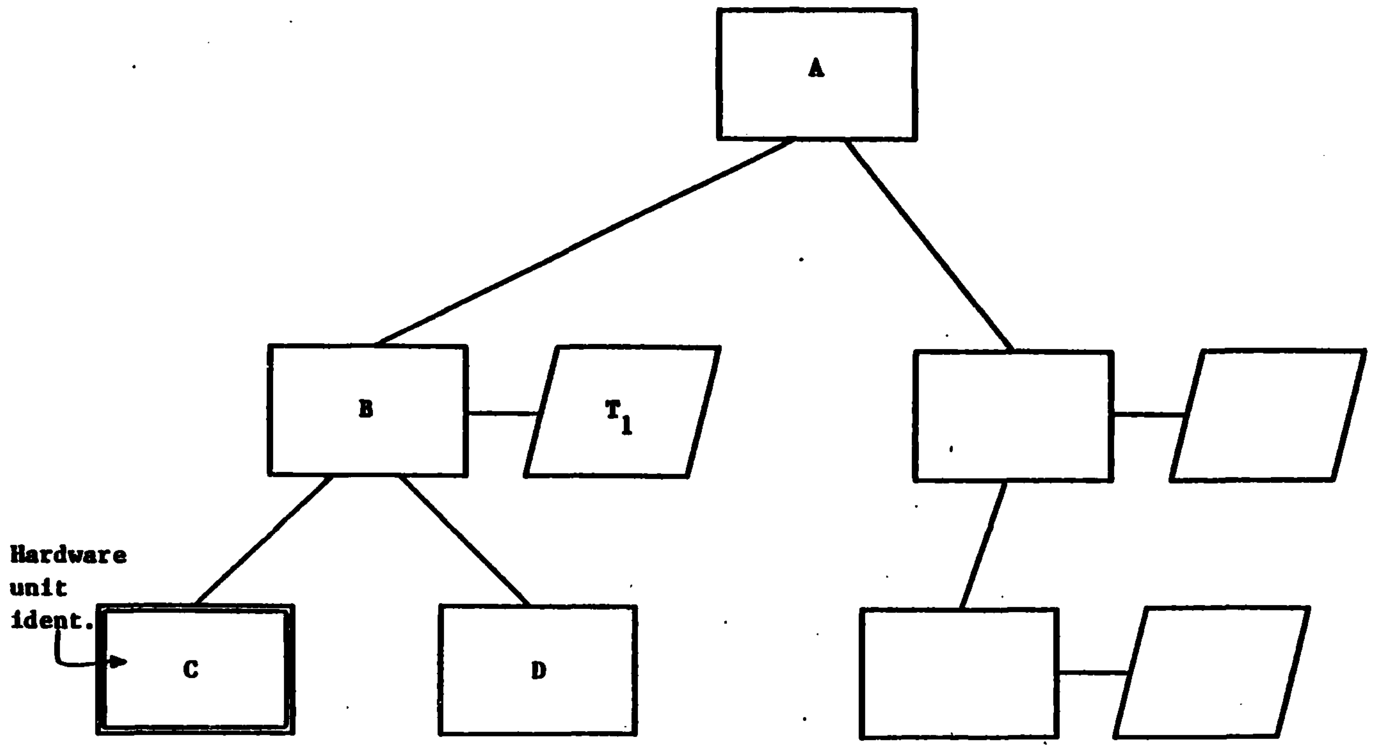


Figure 3.2.10-1 Example of Representation of Hardware Functional Allocation on Structure Chart.

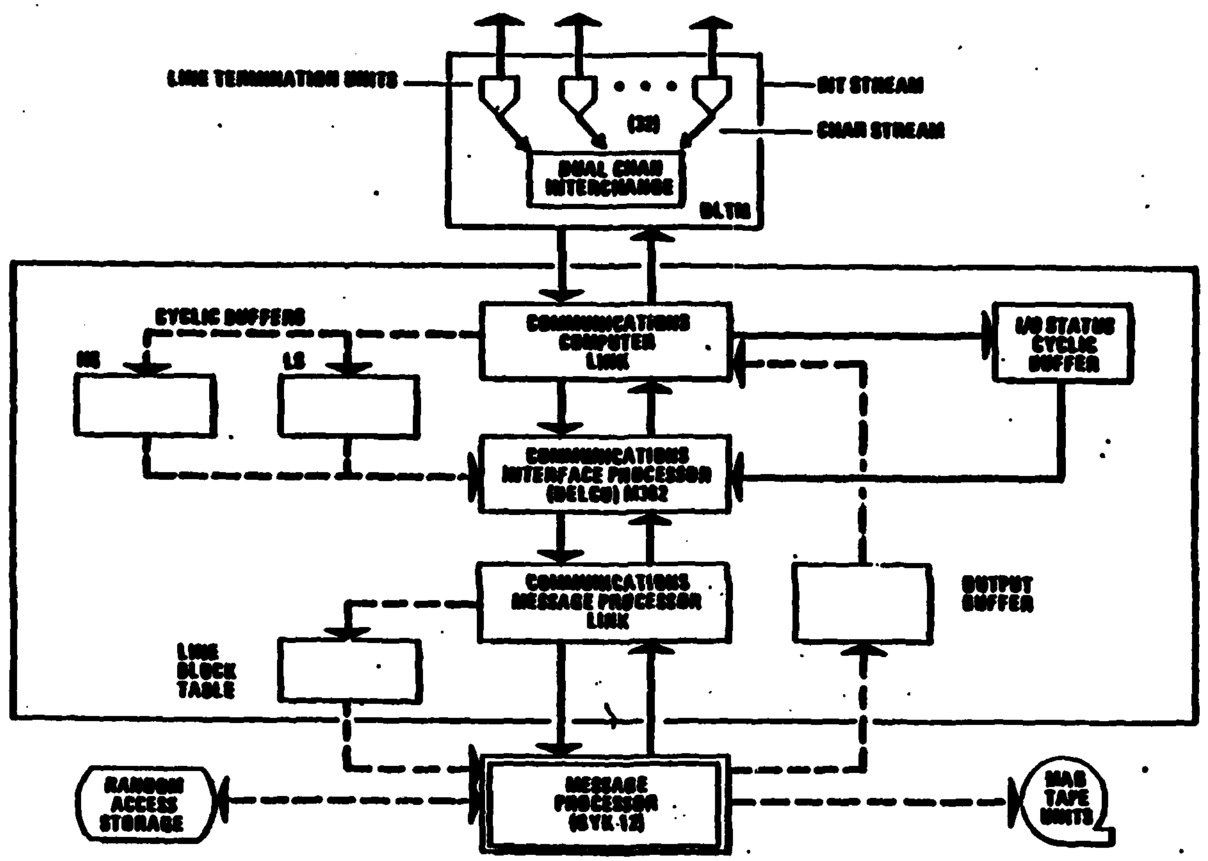


Figure 3.2.10-2 Example of Representation of System Physical Configuration.

3.3 PART II - DETAILED DESIGN

3.3.1 Development and Documentation of Detail Design in Ada/PDL

3.3.1.1 Description

Detail design is an iterative activity which begins with a representation of system architecture. It ends when the designer is satisfied that enough information has been provided to implementation and coding personnel to assure that system functional and performance requirements will be met.

The AIM methodology provides that Ada will be used to document the development of detail design; that is, Ada will be used as a Program (or Process) Design Language (PDL). Inputs to the detail design activity include the following:

a. All graphical representations of system design.

b. Ada unit specifications (declarations) and descriptive commentary, and body stubs, from 3.2.9.

c. System requirements from the A-spec equivalent, via the traceability documentation of 3.2.7.

d. Hardware interface specs from 3.2.10.

3.3.1.2 Guidelines

3.3.1.2.1 Development Activity. Detail design decisions shall be recorded as elaborations of and additions to the architectural design expressed as Ada program units (as described in 3.2.9). Ada constructs shall be used to express design features through successive stages of refinement, in terms of module and system interface specs, variable and aggregate type, range, or enumeration specification, selection and decision logic, data definition, and algorithm functional specification.

3.3.1.2.2 Format. The general form of an Ada/PDL unit shall be as follows:

a. program unit specification;

b. prologue (author, date, functional description, revision level); NOTE: if the unit is a body that has a corresponding unit specification produced as described in 3.2.9, the revision level of that unit spec shall be prefixed to the revision level of the body;

c. unit design in terms of Ada constructs.

Any feature or construct of the Ada language may be used. Program constructs shall be balanced but otherwise are not required to be compilable. Logical commentary may be used freely to describe program unit function or subfunction within a unit. "Action" commentary (preceded by "-->") may be used to indicate incomplete specification at intermediate stages of detail design. Indentation should be made to indicate a nested construct; symmetrical keywords (such as if - endif) should be at the same level of indentation, and statements within larger constructs shall begin at a further level of indentation than that of the enclosing construct.

3.3.1.3 Examples

The following pages illustrate the use of Ada/PDL at an intermediate level of detail design.

Example (1) (due to Grady Booch):

```
procedure TEXT is

--   Body author, date, revision level (rev. level of
--   specification part followed by rev. level of body)
--   Traceability references
--   Short description of unit function, e.g., "Process Text"

begin

          --> ignore leading blanks
     if   line to be centered   then
          --> align text
          --> put out line
     elsif   line is blank   then
          --> put out line
     elsif   not in fill mode   then
          --> put out line
     else -- handle word-by-word
          loop
               --> get a word
          exit when no_more_words
               --> put out word
          end loop;
     end if;
end TEXT;
```

Example (2) (due to Hal Hart):

```
with GLOBAL.PACKET, OTHER_GLOBALS

procedure FORMATTER is

-- (Author, Date, Revision, Description)

    while MORE PACKETS IN FORMATTER loop
         CHK_FORMAT;
         SEND_PACKER;
    endloop;
end FORMATTER;

procedure CHK_FORMAT is

    function VALID (HEADER) return boolean
    procedure READER (PACKER:  out)
    function body VALID is separate
    procedure body READER is separate
         •
         •
         •
begin CHK_FORMAT
    READER (PACKET);
    if VALID (HEADER) then
         if SEC_LEV > TWO_LEV'FIRST then
              case DISPO is
                   when MSG => --> Reassemble to forward
                   when CTL => --> Update control tables
                   when NOTES => --> Copy to log file
              end case;
         else Signal low security level
    end if;

exception when CASE_ERROR => Signal operator
end CHK_FORMAT;
```

3.3.2 Final Design Review

3.3.2.1 Description

The Final Design Review is, as its name implies, the last step of the design process. Therefore, it is mandatory that every reasonable effort be made to ensure that the design developed is an acceptable solution to the problem at hand.

A set of reviews is recommended for the Final Design Review. The set consists of (1) a walk-through of the design, (2) a preprogramming Ada evaluation (3) another review of traceability from requirements to design, and (4) a design philosophy review.

3.3.2.1.1 Design Walk-Through. The design walk-through is a manual simulation of the system designed. Some sample inputs are vicariously processed through the system as designed on paper.

3.3.2.1.2 Preprogramming Ada Evaluation. There are several evaluations that should be made by the designer prior to submitting his product to the programmer. One is the uniqueness versus overloading of names or identifiers which he has defined in the design. Ada does support overloading, and overloading can be used very advantageously such as the case of subprograms which accomplish identical functions but on different structures or types. All overloading should be evaluated to insure that it is intentional and that it will not cause confusion or conflict in any of the places in which it may be referenced.

Data types should be evaluated to insure that duplicate types have not been introduced inadvertently, and that a type is not being used for several purposes which may conflict or permit undetected errors. An evaluation of data type range constraints should be made in relation to performance. Most checking concerning types is done during compilation. However, range constraints may necessitate runtime checks, and depending on the particular compiler being utilized and the particular ranges, may introduce undesirable overhead.

In multi-tasking designs the allocation of entries to specific tasks should be carefully evaluated. Ada supports a variety of strategies for interprocess communication. Careful evaluation must be made to insure that the proper strategy has been selected for each instance of interprocess communication. A good discussion of interprocess communication in Ada is found in "Tutorial on Ada Tasking" by Stephen A. Schuman.

3.3.2.1.3 Requirements-to-Design Traceability. A description of Requirements-to-Design Traceability is provided in step 3.2.7 of architectural design.

3.3.2.1.4 Design Philosophy Review. Oftentimes the original designers of a system are not available to assist with maintenance changes. Therefore, the programmers and analysts that maintain a system in production are often at a disadvantage.

This is particularly true when there is neither sufficient documentation nor expertise available to maintenance personnel. Even when system documentation is available, it does not always provide the maintenance personnel with enough information about the design philosophy.

During the design philosophy review, the designer(s) should review the overall design and document the design philosophy that was utilized to make major design decisions. This will result in a Design Philosophy Document to be used and maintained after the system is in production.

3.3.2.1.5 Iterative Design Process. The designer should correct any problems identified by the review process above. Appropriate steps within the entire design should be repeated as required to correct the problem(s) identified.

3.3.2.2 Guidelines

3.3.2.2.1 Design Walk-Through.

a. Invite all the designers to the walk-through.

b. Designate a chairperson to preside over the walk-through and resolve differences.

c. Conduct the walk-through in a place as free from disturbances and interruptions as possible.

d. Consider inviting a user/customer representative that has a good understanding of the problem being solved by the design.

e. Select some good sample inputs to be vicariously processed (walked) through the system. The inputs selected should (as a whole) exhibit characteristics that will test each function of the system.

f. Develop desired outputs for each input to be walked through the system.

g. Document the problems encountered as inputs are walked through the system.

h. Follow up on each problem discovered in subsequent walk-throughs.

3.3.2.2.2 Preprogramming Ada Evaluation.

a. Obtain a complete cross reference list of all identifiers or names utilized in the design.

b. Examine each instance of overloading to prevent ambiguous references or any possible conflict or confusion.

c. Examine the uses of each data type to insure that sufficient types have been introduced and that no types have been inadvertently duplicated.

d. In light of performance requirements, evaluate data types for runtime overhead they may introduce.

e. When collections of data are being passed as parameters, consideration should be given to using records containing the collections and passing access values to indicate the record. This can reduce parameter passing overhead when carefully done.

f. In multi-tasking, the preferred method of protecting critical code is within the "do...end" of a rendezvous, so as to be clearly evident during maintenance.

However, in a few cases in a multiprocessor environment, when there is a considerable amount of critical code and when it is desirable to maintain two threads of execution for performance, a scheme whereby the critical code is placed between two different rendezvous may be used. These cases must be clearly commented, as protection is only supplied by programmer convention and must not be violated during maintenance.

g. In multi-tasking, evaluate all accesses to shared data to insure that all accesses are properly synchronized and that all critical code is properly protected by in an appropriate rendezvous.

h. Evaluate all entry calls and accept statements to insure that an appropriate strategy has been properly used for each interprocess communication.

i. When task activation and termination is used in a system, a careful review must be made of task dependencies to ensure that deadlock will not occur.

3.3.2.2.3 Requirements-to-Design Traceability. The guidelines for traceability are provided in step 3.2.7.2.

An example of a requirements-to-design traceability matrix is provided in step 3.2.3. A suggested outline for the Design Philosophy Document is provided below.

3.3.2.2.4 Design Philosophy Review.

a. Document the philosophy behind major design decisions. Specifically, give reasons for the following:

   - Overall structure of system
   - Ada packaging
   - Concurrency/Tasking
   - Hardware selections
   - Data Structures (including files, data bases, stacks, queues, lists, etc.).

b. Document strategies used and how they impacted design (e.g., transform analysis, transaction analysis, etc.).

c. Develop a list of design admonitions (e.g., strength of design, weakness of design, static design areas, dynamic design areas, and suggestions for improvement).

d. Integrate all design philosophy documentation into a Design Philosophy Document.

e. Follow up on any problems that might arise as a result of this final review of the design philosophy.

DESIGN PHILOSOPHY DOCUMENT

I. SYSTEM STRUCTURE

  A. OVERALL STRUCTURE

    1. FUNCTIONAL DECOMPOSITION
    2. HIERARCHY
    3. FUNCTIONAL HARDWARE COMPONENTS

  B. ADA PACKAGES

    1. OBJECTS AROUND WHICH PACKAGES WERE FORMED
    2. REASONS FOR FORMING PACKAGES

  C. CONCURRENCY/TASKING

    1. CONCURRENCY REQUIRED SPECIFICALLY BY THE CUSTOMER
    2. CONCURRENCY INCORPORATED INTO DESIGN FOR SIMPLICITY OR PERFORMANCE REASONS
    3. TASKS
    4. DEADLOCK CONSIDERATIONS
    5. SHARED RESOURCE/VARIABLE CONSIDERATIONS
    6. HARDWARE SUPPORTING CONCURRENCY

  D. DATA STRUCTURES

    1. PERMANENT DATA STRUCTURES

      a. DATA BASES
      b. FILES
      c. TABLES

    2. TEMPORARY DATA STRUCTURES

      a. STACKS
      b. QUEUES
      c. LISTS

II. DESIGN STRATEGIES

  A. TRANSFORM ANALYSIS

  B. TRANSACTION ANALYSIS

  C. OBJECT-ORIENTED DESIGN

III. DESIGN ADMONITIONS

  A. STRENGTH OF DESIGN

  B. WEAKNESS OF DESIGN

  C. AREAS OF SYSTEM MOST LIKELY TO CHANGE

  D. AREAS OF SYSTEM LEAST LIKELY TO CHANGE

  E. SUGGESTIONS FOR IMPROVING DESIGN

Figure 3.3.2-1 Suggested outline for Design Philosophy Document

3.4 PART III - ADA DESIGN DOCUMENT

3.4.1 Description

The output of design must be organized into an Ada Design Document. This document will be given to the implementation team when programming begins.

3.4.2 Guidelines

Organize the Ada Design Document according to the outline below:

I. Design Philosophy Document

II. Design Graphic Illustrations

A. System Chart
B. Structure Chart
C. Packages Chart
D. N^2 Chart
E. Logical Data Structures Model
F. Jackson Data Structures
G. Requirements-toDesign Traceability Matrix

III. Ada Representation of System.

CHAPTER 4

Ada DEVELOPMENT STANDARDS

4.1 INTRODUCTION

Language proliferation has been one of the DoD's most significant software problems. Custom languages and compilers have been developed "on the fly" for individual projects. Often, the languages and compilers have been developed hastily. This has resulted in project failures and unsatisfactory results.

Short-term and long-term solutions to the language proliferation problem have been developed. The short-term solution was the establishment of a set of seven programming languages to be used for the development of all DoD software. Unfortunately, the seven languages of the standard set do not have all of the features needed for the development of DoD software.

The long-term solution was determined to be a conversion to one common programming language. Benefits from conversion to a single common language were estimated at over $100 million per year. The DoD's requirements for one common language were expressed in the Steelman document. In meeting the Steelman requirements, the designers of Ada have created a large and complex language with powerful constructs and semantic subtleties that offer ample opportunities for failures as well as successes.

Although Ada was initially intended to be the implementation vehicle for embedded real-time systems, current experience and interest indicate that the language will be applied to a much wider problem domain. We expect that Ada will be used in scientific and business applications in addition to the expected applications in avionics, communications, and command/control. Further, there is interest in Ada as a requirements and design language and as a hardware description language for designers of digital systems. In short, Ada is seen by many as a potential "universal language" for software engineering.

Ada's newness, size, complexity, and global appeal make it essential that a good set of Ada development standards be implemented and enforced. Unless Ada devlopment standards are applied, designers and programmers with varying backgrounds will be developing Ada systems with many variations of style and maintainability. Therefore, an initial set of Ada Development Standards has been created as a part of the effort associated with the Ada Capability Study contract. The standards developed have been categorized as follows:

A. Template for Ada Compilation Units

- Design
- Programming

B. Preferred Programming Practices

- Documentation
- Basic Constructs
- Declarations
- Maintainability
- Naming Conventions
- General Coding Conventions

One important category not listed above, for which standardization is essential, is portability. The article, Ada-European Guidelines for the Portability of Ada Programs, by Nissen, Wallis, Wichman, and others, that appeared in the March,April 1982 issue of Ada letters, provides a set of recommended standards for Ada portability.

4.2 STANDARDS

Ada development standards address both the design and implementation phases of system development. The "Template for Ada Compilation Units" section includes design and programming standards. All other sections consist of only programming standards.

There are two types of Ada development standards--requirement and guideline. Each type is defined below:

Requirement	- requires complete adherence by the designer/programmer and shall be enforced by the project manager
Guideline	- suggests adherence but deviation is permitted when designer/programmer can show proper justification.

Maintainability is the theme that permeated the creation of the Ada Development Standards; performance was not a consideration. However, realizing that Ada will be used in a real-time, embedded systems environment, there may be cases where the project manager will choose to relax adherence to the standards for performance reasons. It is suggested that these compromises be made only after hard evidence supporting them is established and documented.

4.2.1 Template for Ada Compilation Units

The maintainability of a program is directly related to its structure. Ada provides some program structuring capabilities that may be used advantageously or detrimentally. The purpose of the standards in this section is to direct the developers (designers and programmers) in their use of (1) the Ada structuring capabilities and (2) basic design heuristics.

4.2.1.1 Template Design Standards

The Ada designer must be familiar with the Ada concepts of subprogram, packaging, nesting, and local variables/visibility. Additionally, the Ada designer should be familiar with some of the basic design heuristics that facilitate development of good maintainable systems (regardless of the implementation language). This subsection provides some design standards that will assist the designer in structuring a program for maintainability.

4.2.1.1.1 Control Routine/Package Services. It is recommended that Ada systems be designed according to the principles of object-oriented design and information hiding. Following these principles will result in a design that encapsulates data structures and all their operations (services) within Ada packages.

The design must integrate the package services within a total system. Ideally, the package services will be called as needed by control routines within the system hierarchy. The control routines are the subprograms responsible for accomplishing and coordinating the activities of a given function. The advantages expected from the application of this concept are:

A. Simplicity - derived from hiding data structures and design decisions.

B. Clean interfaces to black boxes (package services).

C. Well-partitioned, highly factored systems.

D. Limited nesting and visibility problems.

E. Support of maintainability.

In order to achieve these advantages, the following guidelines have been established.

Guideline	- Develop Ada packages that provide services for a given data structure.
Guideline	- Use the information hiding concept to hide data structures and design decisions - provide services that conceal the data structures and internal details from the calling subprogram.
Guideline	- Use Private/Limited Private constraint on data structures to support information hiding and abstract data types during design.
Guideline	- Use Limited Private constraint when comparisons and/or assignments of data could produce inaccurate results.

4.2.1.1.2 Limited Nesting. Methodical application of structured programming techniques may lead to the creation of programs with many levels of nested subprograms. Extensive nesting tends to reduce the benefits of methodology in the maintenance phase of the software life cycle. The article, Nesting in Ada Programs is for the Birds, by Clark, Wileden,and Wolf, that appeared in the November 1980 issue of Sigplan Notices, outlines these problems with respect to Ada and proposes a means to completely avoid nesting. However, we believe a disciplined utilization of nesting should be available to the software designer.

Guideline	- Avoid subprogram nesting whenever possible; exploit packages and separate compilation to enhance readability and maintainability.
Requirement	- Subprogram nesting shall be limited to at most three levels except as approved on a case-by-

case basis by the manager responsible for the project.

4.2.1.1.3 Local Variables and Visibility. The structured programming purist imposes severe constraints on access to data within each subprogram. All variables accessed should be either local to the subprogram or parameters to the routine. For embedded systems design, this may be rather inconvenient; for example, access to several global state variables is often a necessity.

Guideline	- Avoid references to global variables. If such references are necessary, the variable references should be qualified by the name of the routine in which the variable is actually declared, so that the global reference is explicit.
Requirement	- Each variable shall be declared locally to the scope of its use in order to minimize introduction of errors by erroneous modification of data.

4.2.1.1.4 Single Entry/Single Exit. The single entry/single exit concept is a pre-Ada structured programming concept that supports simplicity, consistency, and maintainability of code.

Requirement	- All Ada subprograms shall have one entry and one exit.

4.2.1.1.5 Size. The size of each Ada design unit is important in terms of maintainability. Studies have shown that persons can cope with multiple small problems easier and more efficiently than they can cope with one large problem. Considering this, the size of each Ada design unit body should be limited to a reasonable number of executable statements.

Guideline	- Each Ada design unit body should not exceed 200 lines of executable code.

4.2.1.2 Template Programming Standards

This subsection includes programming standards to be applied during the implementation phase. These standards support the development of good, maintainable programs and the implementation of development strategies and/or test plans.

4.2.1.2.1 Use of "is separate"

Guideline	- Use "is separate" to defer the implementation of lower level details in the top-down development of a software system.
Guideline	- The use of "is separate" should be well documented if the separate compilation unit is

sensitive to global data (within the context of nesting).

4.2.1.2.2 Use of "with"

Requirement - Each routine shall have access only to packages required for implementation of the routine.

4.2.1.2.3 Order of Declarations

A uniform approach to declarations of objects in Ada is necessary to promote readability and ease of maintenance.

Requirement - All declarations possessing the same characteristics shall be grouped together and commented accordingly. For example, all CONSTANTS shall be together, all TYPES together, etc. Accordingly, all SUBTYPES shall immediately follow (and optionally be indented from) their respective TYPES. A blank line should be used to separate the groups.

Requirement - If constants of a user-defined type are required they should be grouped together into a separate CONSTANT declaration immediately following the TYPE declarations.

Requirement - Within each group named above (except TYPES), all identifiers should be arranged alphabetically by name. (This is difficult to demand for TYPES, for which meaningful names and elaboration order requirements may preclude alphabetic arrangement.)

```
-- CONSTANT DECLARATIONS

    BUFFER_SIZE :  constant := 100 ;

    PI          :  constant := 3.1416 ;

    RECORD_SIZE :  constant := 80 ;

-- TYPE DECLARATIONS

    type WEEK_DAYS is ( MONDAY, TUESDAY, WEDNESDAY,THURSDAY,
                        FRIDAY, SATURDAY, SUNDAY ) ;

    type MONTH_NAME is ( JAN, FEB, MAR, APR, MAY, JUN,
                         JUL, AUG, SEP, OCT, NOV, DEC ) ;

      subtype FIRST_QTR is MONTH_NAME range JAN .. MAR ;

    type DATE is
      record
        MONTH   : MONTH_NAME ;
        DAY     : INTEGER range 1 .. 31 ;
        YEAR    : INTEGER range 1900 .. 1999 ;
      end record ;

-- CONSTANTS OF USER-DEFINED TYPES

    FIRST_DAY : constant WEEK_DAYS := MONDAY ;
    LAST_DAY  : constant WEEK_DAYS := SUNDAY ;

-- VARIABLES

    ANNIVERSARY : DATE ;
    ANSWER      : BOOLEAN := FALSE ;    -- INIT. IN DECLARATION
    HOFL        : INTEGER ;             -- HEAD OF FREE LIST
    RECORD_TYPE : INTEGER ;
```

Figure 4.2.1 Order of Declarations Example

4.2.2 Preferred Programming Practices

The enforcement of a good set of programming standards will foster understanding and consistency of the code developed. Understanding and consistency of the code will contribute to maintainability, the salient theme behind the development of the standards.

4.2.2.1 Documentation

4.2.2.1.1 Preamble Documentation

Requirement - Every procedure, function, or package shall include a preamble placed between the name declaration and the source code.

Requirement - Every preamble shall include at least the following items:

* Name of Unit
* Abstract--short description of purpose
* Processing description--algorithms/literature references
* What errors are handled and how--exception/status returns
* Definitions of all input and output variables

Requirement - When a particular hardware or software configuration for compilation and linking (host processor) or execution (target processor) is required by a package or routine, that dependency must be detailed in the preamble.

```
procedure GCD ( X,
                Y               : in INTEGER ;
                COMFACTOR       : out INTEGER ) is

-- *****************************************************************
--      Mnemonic : GCD
--      Usage    : compute greatest common factor of two integers
--      Author   : Euclid
--      Host
--         machine : Cray-4
--         o/s     : CP/M
--      Target
--         machine : Intel 80286
--         o/s     : MP/M 286
--      Abstract :
--         Given two integers X and Y as input, the greatest common
--         factor of X and Y is computed and returned to the caller
--         through the formal out parameter COMFACTOR.
--      Processing :
--         << description of algorithm >>
--
--         << error conditions with exceptions and messages >>
-- *****************************************************************

                  << body of procedure here >>
```

Figure 4.2.2-1 Preamble Example

4.2.2.1.2 Comments

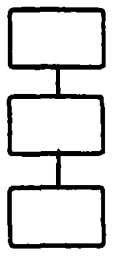
Requirement - Comments associated with structured statements shall align.

Requirement - Comments appended at the end of a line and continued to successive lines shall be written so the continued comment fragments physically align under the beginning comment fragment. Subsequent statements should not begin until after the comment has been completed.

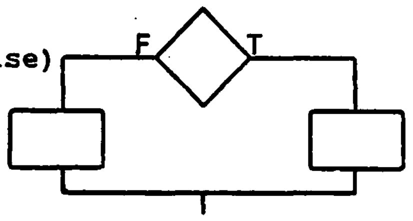
4.2.2.2 Basic Constructs

Bohm and Jacopini proved that all computer programs may be written using only three basic constructs. These three constructs are described on the following page using flowchart symbology.

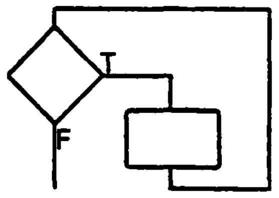
sequence



selection
(if-then-else)



iteration
(do while)



Of course, Ada supports these three constructs. Ada also supports extensions to the selection and iteration constructs. The selection is supported in Ada by the "if-then", "if-then-else", and "case" constructs. The iteration ("do while") is supported by the "while/for condition loop" construct in Ada. An infinite iteration is simply supported by the "loop" (without the "while condition") construct. A "do until" iteration, in which the action is performed within the loop before the condition check, is supported in Ada by a combination of the loop and exit constructs (exit must be placed inside loop to be executed when a condition is true).

The sequence construct is general in nature and will not be addressed specifically in the standards below. However, standards have been developed for the selection and iteration constructs and their Ada extensions.

4.2.2.2.1 Selection Standards

Requirement - The if-then construct shall only be used when there is an action associated with the true condition and no action associated with the false condition.

Note: The if-then-else construct may also be used when there is only an action associated with a true condition by placing "null" after the else.

Guideline - Nested if's are preferred over compound complex single if statements.

Requirement - There shall be at most three levels of nested if's.

Guideline - The case statement is preferred over a series of if statements whenever appropriate.

Requirement - Statements associated with the if statement must align and be indented consistently, but by at least two spaces. This standard is illustrated by Figure 4.2.2-2.

Requirement - The 'then' associated with the true condition of an if statement must be coded on the same line as the 'if'. The action(s) associated with a true condition must start on the line immediately below the if. Figure 4.2.2-2 illustrates this standard.

Requirement - The statement elements associated with the case statement shall align, and be indented consistenly, but by at least two (2) spaces. Figure 4.2.2-3 illustrates this requirement.

4.2.2.2.2 Iteration

Requirement - The "do while" iteration shall be coded using the 'for' or 'while' condition with the 'loop' verb. (See Figure 4.2.2-4.)

Requirement - Statements associated with the loop statement and all variants of the loop statement shall be indented consistently, but by at least two spaces. This standard is illustrated by Figure 4.2.2-4.

Requirement - The 'do until' iteration (action performed before condition check) shall be implemented with a loop statement and an 'exit when' statement within the loop. This standard is illustrated by Figure 4.2.2-5.

Requirement - Avoid using multiple exit statements in a loop.

Requirement - The infinite loop shall be implemented with a loop statement without any conditions. (See Figure 4.2.2-6.)

```
if INDENT then
   CHECK_LEFT_MARGIN;
elsif OUTDENT then
   RIGHT_SHIFT;
else
   CARRIAGE_RETURN;
   CONTINUE_SCAN;
endif;
```

Figure 4.2.2-2 Example of IF Statement Indentation

```
case TODAY is
  when MONDAY =>
    COMPUTE_INITIAL_BALANCE ;
  when TUESDAY .. THURSDAY =>
    GENERATE_REPORT(TODAY) ;
  when FRIDAY =>
    COMPUTE_CLOSING_BALANCE ;
  when SATURDAY .. SUNDAY =>
    null ;
end case ;
```

Figure 4.2.2-3 Case Example

```
for I in 1 .. 10 loop
  SUM := SUM + TABLE(I);
end loop;

INDEX := 1
while INDEX <= 20 loop
  if TABLE(INDEX) > MAX then
    MAX := TABLE(INDEX);
  end if
  INDEX := INDEX + 1;
end loop
```

Figure 4.2.2-4 Ada "Do While" Loops

```
INDEX := COUNT
loop
  if BRIDGE(INDEX) > MAX then
    MAX := BRIDGE(INDEX);
  endif;
  INDEX := INDEX + 1;
  exit when INDEX ≥ 20;
end loop;
```

Figure 4.2.2-5 Ada "Do Until" Loop

```
PERPETUAL_LOOP :        --note indentation
  loop                  --from loop identifier
    COUNT := COUNT + 1;
    GET(TEMP);
    PUT(TEMP);
  end loop;
```

Figure 4.2.2-6 Ada Infinite Loop

4.2.2.3 Declarations

This subsection specifies those Ada coding practices which apply to the use of declarations.

4.2.2.3.1 Commenting Declarations

Requirement - Each constant, type variable, and field identifier declared shall be accompanied by a brief COMMENT if the mnemonic identifiers are not self-explanatory.

4.2.2.3.2 Program Messages

Requirement - All coded program messages shall be declared as program constants and the messages shall be identified in the "OUTPUT" or "ERROR CONDITIONS" section of the preamble.

4.2.2.3.3 Declaration Formatting

Requirement - Every declaration shall begin on a new line and shall be aligned with all preceding declarations.

Guideline - It is recommended that each variable of a declaration appear on a separate line.

Example:
```
X,
Y : INTEGER
```

Guideline - All constants and variables should be an instance of a named type; in particular, array and record objects should reference explicit type names.

4.2.2.3.4 Use of Constants

Requirement - All SCALARS that (1) are used in more than one instance within the same context, (2) are known scientific or engineering constants, or (3) have associated dimensions such as buffer size, terminal width, etc. that are susceptible to change shall be specified as named program constants.

Guideline - The upper bounds of all static arrays should be specified by program constants.

4.2.2.4 Maintainability

Guideline - Comments should be composed at the same time that code is composed.

Guideline - Functions should avoid side effects. For example, the value of a global variable should not be changed. Any exceptions should be conspicuously documented.

Guideline - All type declarations should include bounds that constrain variables to the expected range of values. For example, if the range of an integer-valued type is known an appropriate type declaration should include those constraints:

```
type TWENTIETH_CENTURY = 1900 .. 1999 ;
```

Guideline - Separate types should be declared specific to their use. The type-checking capability of Ada may be exploited if new types are declared for each use, although some types may overlap.

```
type ORANGE_CNTY is new NATURAL ;
type PECAN_CNTY is new NATURAL ;
```

Requirement - Each statement shall begin on a separate line.

Requirement - At least one space shall appear before and after all relational operators, Ada reserved words, identifiers, and arithmetic operators.

Requirement - Use explicit assignments in aggregates in lieu of positional assignments.

Guideline - Use attributes instead of hard coding numeric literals whenever possible.

Guideline - Range constraints on arrays should be specified using named constants or attributes of enumeration types.

Guideline - Use parentheses in expressions to enhance clarity.

Guideline - Write code with lots of white space to enhance readability. Avoid bunching variable names, relational operators, subscripts together.

4.2.2.5 Naming Conventions

Guideline - Use meaningful names for all types, objects, subprograms, and tasks.

Requirement - A consistent naming convention shall be established for all objects of interest at the outset of the project. This should form the basis for the project data dictionary.

Requirement - The use of overloaded names for subprograms shall be restricted to those instances in which subprograms perform semantically identical operations on differing data types.

Guideline - If a task has a single entry point, it is recommended that this entry be named "ENTRY_POINT", or some similar name; this avoids the introduction of awkward names in such contexts.

Guideline - Use some naming convention to distinguish access types from other data and routine names which may be used as "qualifiers". For example, if access variable names are given a consistent suffix "_ptr", then all access variable references are textually emphasized.

Guideline - The names of types and constants should reflect their use. A value used in more than one context should have an equivalent set of names and declarations. An example is provided below.

```
NUMBER_OF_MONTHS : constant := 12 ;
DOZEN            : constant := 12 ;
```

Guideline - The names of types and objects should not be a simple recapitulation of their values. For example, one should avoid declarations such as

```
ZERO             : constant := 0 ;
```

Guideline - Avoid use of the "use clause" when multiple packages are imported and local variables have same names.

4.2.2.6 General Coding Conventions

This subsection specifies general coding practices which apply to the coding of Ada program statements.

4.2.2.6.1 Label and GO TO

Guideline - Label and GO TO statements should be used sparingly. Their use should be restricted or controlled within the boundary of one block. Generally, a GO TO should only branch to the beginning or end of a block from a point within the block. Label and GO TO statements should not be used to transfer control from block to block and should not traverse large segments of code.

4.2.2.6.2 Upper and Lower Case

Requirement - For uniformity, all declared items shall be capitalized and all reserved words shall be in lower case.

4.2.2.6.3 Program Units

Requirement - Each program unit shall be preceded by at least one blank line.

Requirement - The end which concludes a subprogram body, package, specification, package body, task specification, or task body, shall always be followed by the name of the program unit.

Requirement - Statements associated with blocks shall align and be indented consistently, but by at least two spaces. This standard is illustrated by Figure 4.2.2-7.

4.2.2.6.4 Procedure and Function Subprograms

Requirement - If the procedure declaration and associated parameter list does not fit readily on one line, the parameter list shall be aligned and coded with one parameter per line.

Requirement - The begin and end surrounding a subprogram body shall be aligned under the subprogram specification. All other statements shall be indented consistently, but by at least two spaces.

Guideline - Parameters should be grouped together according to their function as either in, out, or inout.

Requirement - In any given call to a subprogram, all parameter references shall be of a consistent type; i.e., positional and keyword parameter references shall not be mixed in a given call to a subprogram.

```
                   SWAP :
                     declare
                       TEMP : INTEGER ;
                     begin
                       TEMP := V;
                       V    := U;
                       U    := TEMP;
                     end SWAP;
```

Figure 4.2.2-7 Example of Block Indentation

```
function FACTORIAL ( N : NATURAL ) return FLOAT is

  TEMP : FLOAT ;

begin
  if N = 1 then
    TEMP := 1.0 ;
  else
    TEMP := FLOAT( N ) * FACTCRIAL( N - 1 ) ;
  end if ;
  return TEMP ;
exception

end FACTORIAL ;
```

Figure 4.2.2.8 Example of Function Standards

```
procedure PUSH ( E : in ELEMENT_TYPE ; S : inout STACK) is
begin
  if S.INDEX = S.SIZE then
    raise STACK_OVERFLOW ;
  else
    S.INDEX := S.INDEX + 1 ;
    S.SPACE(S.INDEX) := E ;
  end if ;
end PUSH ;
```

Figure 4.2.2-9 Example of Procedure Standards

Guideline	- In instances in which formal parameters may be given highly mnemonic names, the use of keyword parameter references should be exploited to enhance readability and ease of maintenance.
Requirement	- Explicitly identify the mode of each parameter in a call (avoid 'in' parameter default).
Requirement	- Avoid the use of default parameters in procedures and functions.

4.2.2.6.5 Exceptions

Requirement	- The exception reserved word shall align under the begin of the appropriate block.
Requirement	- Each exception choice within an exception handler shall align and be indented consistently, but by at least two (2) spaces.

4.3 CONCLUSION

A myriad of new Ada programmers with various backgrounds and varying degrees of expertise is developing. Therefore, the DoD must recognize that Ada programs will differ significantly in style, quality, and maintainability unless Ada development standards are implemented and strictly enforced.

The implementation of Ada development standards must be planned. Procedures for ensuring that standards are followed should be in place when Ada becomes the DoD's official programming language. Additionally, certain tools should be developed in support of the standards.

4.3.1 Tools Needed

No development standard will be successful if it impedes the work of the software engineering professional. One means of easing the individual impact of a standard is to provide aids for the use of such a standard. The following toolset is a partial list of aids which will provide enhanced capabilities for implementation and maintenance personnel. Note also that there is some overlap in the functions suggested, leaving some flexibility to the user.

A. An enhanced cross reference and "use table" generator which provides a means to resolve variable overloading in each subprogram.

B. An "expand" compiler pragma to replace all global name references by fully qualified names in the text of a subprogram.

C. An "eject" pragma to control pagination in the compiler listing for clarity.

D. A "package reference expansion" pragma to list the visible part of a referenced package in the declaration part of the routine in which the reference occurs, with appropriate notation to indicate the expansion.

E. A "pretty print" facility which enforces as much of the development standard as may be automated, and notes any violations detected.

4.3.2 Need for Continued Research and Development

Any standard for Ada development must be closely related to a systems design methodology. Many problems must be solved with relation to the methodology such as a "graceful" incorporation of the Ada tasking model, which will have an obvious impact on the development standards. The standard itself must be refined in light of experience. As the Ada experience base grows, new issues relating to standards will arise and must be addressed as they occur. Also, the impact of the standard on programmer productivity and maintainability must be assessed. Some restrictions currently in the proposed standard may be relaxed once a sufficient population of Ada-literate programmers and designers exists. Significant work remains in the area of standards relating to the timely introduction of architectural dependencies in the design process.

4.3.3 Importance of Ada Development Standards

The newness of Ada combined with its power and the shortage of appropriately trained personnel make the existence of a development standard a practical necessity. The standard presented has been created as a "first order approximation", restricting the use of certain novel features of Ada such as overloading and parameter associations to modes more closely resembling languages in current use. On the other hand, there is little said in regard to exceptions and tasking as effective use of these features requires some empirical evidence to support creation of reasonable standards. This standard should be viewed as the beginning of an evolutionary process which will be driven by the accumulated experience in the use of Ada as well as growth in the population of Ada language users. It is most important that a variant of this standard or some other standard be in place before the availability of validated compilers so that a standard may develop in light of a large body of experience. This would be of benefit to developers and consumers of Ada-based software.

CHAPTER 5

METHODOLOGY EXPERIENCES AND RECOMMENDATIONS

5.1 PREFACE

The development of AIM and its subsequent application to the redesign of a message switch system has been a very beneficial, learning experience. There have been many lessons learned as well as suggestions and ideas for improving AIM. Three activities that have been most beneficial in terms of evaluating and improving AIM are briefly described below:

1. Initial Application of AIM - As soon as a phase of AIM was completed, it was applied by the design team. This resulted in the discovery of "bugs" in AIM that were immediately corrected. However, some problems identified were not immediately solvable. These problems contributed to the lessons learned on this project.

2. Consultants' Suggestions and Ideas - Consultants from industry and academia evaluated AIM. Their evaluations resulted in many suggestions and ideas for improvement. Some of these ideas and suggestions have already been incorporated into AIM while others are addressed later in this chapter as recommendations for improvement and recommendations for further research.

3. Continued Methodology Study by Chief Methodology Engineer - The chief methodology engineer has continued to study methodologies and make modifications to AIM after its completion. Some of the lessons learned and recommendations addressed later in this chapter evolved from this continued study.

The remainder of Chapter 5 addresses (1) the lessons learned from the Ada Capability Study, (2) recommendations for improving AIM, and (3) recommendations for further research.

5.2 LESSONS LEARNED

Summaries of the lessons learned from the Ada Capability Study are provided below.

5.2.1 Importance of Using a Methodology

Methodologies provide a plan or road map for system development. Without a plan, the system development of any major system is more susceptible to failure. A methodology, even though it may only be a framework such as AIM, forces the project team personnel to think about the problem and solution in an organized manner. In the words of one of our cons 'tants, "Any methodology is ninety percent good".

5.2.2 Importance of Understanding the Problem

It is extremely important to develop an understanding of the problem in the requirements phase of system development. This is true regardless of the type of system being developed (i.e., business, real-time, scientific, etc.). Once a good understanding of the problem is developed, the design process is ready to begin.

The design team used much more time and effort than originally anticipated to complete the requirements phase. However, the feeling is that the extra time was well spent. The requirements analysts developed an understanding of the problem that reduced the effort required during the design phase.

5.2.3 Use of Ada as an RSL Reduces Design and Programming Efforts

The Ada Requirements Specifications produced during the requirements phase eliminated the need for an Ada PDL expression of the system during the design process. The design team felt that the Ada Requirements Specifications were sufficient specifications to begin program development once architectural design was completed. The programming effort was also reduced because the frameworks of many of the Ada procedures were established during the requirements phase.

5.2.4 Lack of Integration Between Structured Analysis and Structured Design

Structured Analysis and Structured Design methodologies do not integrate as smoothly from requirements to design for real-time communications systems as they dc in a business applications environment.

5.2.5 Ada Can Be Used Throughcut the System Development Life Cycle

Ada has the constructs for forming the base of a structured English for expressing system requirements and programming specifications. Therefore, Ada may be used as an RSL and PDL prior to its use as an implementation language. The use of Ada throughout the system development life cycle reduces the conversion efforts normally required tc map requirements into design and design into code.

5.2.6 Ada is Most Compatible with the Object-Oriented Design Methodology

Ada will support virtually any methodology. However, it appears that Ada is more compatible with the object-oriented design methodology than other methodologies such as Structured Design, Jackson, and Warnier-Orr.

5.2.7 Backgrounds of Personnel Developing Ada Systems is Important

Of course, it is always important to match people with appropriate backgrounds to development projects. The best programmer is not always the best requirements analyst or designer. Therefore, the personnel assigned to the requirements and design phases need not be expert Ada programmers. However, the requirements analysts and designers need to have some knowledge of Ada.

The requirements analysts should be familiar with the basic Ada constructs if Ada is used as an RSL during the requirements phase. Designers should know enough about Ada to use it as a PDL. Additionally, designers should have a good understanding of Ada program structure (i.e., subprograms and packages) and tasking.

5.2.8 Need for More Customer-Oriented Requirements Specifications

ARM is heavily oriented toward Structured Analysis. The Ada language is also key to ARM since it is used to express the system requirements. Therefore, the Ada Requirements Document is largely DFDs and Ada Requirements Specifications. This format is great for the requirements analysts and designers. However, from a customer's point of view the Ada Requirements Document is not a good, clear expression of the system. The Structured Analysis and Ada expressions of the system need to be augmented with more customer-oriented system requirements.

5.2.9 Value of Graphic Illustrations

DFDs and structure charts are good tools that facilitate developing an understanding of the problem and solution during the requirements and design phases respectively. However, the use of such tools puts Ada in a more supportive role than originally anticipated. The graphic illustrations seem to be easier to understand initially than a pure Ada expression of the problem and solution.

5.2.10 Structure Charts are not Designed to Support Recursion

Structure charts have been used prolificly during the design of systems to be implemented in COBOL or FORTRAN. Since neither of these languages is recursive, there is no need for representing recursion on the structure chart. The lack of a structure chart recursion mechanism may be a problem when trying to model an Ada solution that is recursive.

5.2.11 SREM-type Concurrency Charts are not Appropriate for Representing the Concurrency of the Message Switch System

The design team was unable to use the SREM-type concurrency charts to represent the concurrency of the message switch system.

One of our consultants tried extensively to draw a concurrency chart for the message switch system but was unsuccessful. A second consultant indicated that it was impossible to illustrate the concurrency of the message switch using SREM-type concurrency charts.

5.2.12 Automated Design Aids Could Improve the System Development Process

Automated design aids could be used to improve project management, increase productivity, and improve the accuracy of requirements and design specifications. Specifically, automated design aids could be used to do the following:

- Draw DFDs
- Develop traceability matrices
- Structure requirements and design specifications for clarity
- Develop a data dictionary
- Verify requirements and design specifications.

5.2.13 Difficulties Associated with Late Hardware/Software Partitioning (HSP)

HSP is the last step of architectural design (after completion of a functional design). However, experience with the redesign of the message switch system indicates there are difficulties associated with HSP this late in the design process. For example, in a distributed processing environment, the selection of separate processors that use separate memories could significantly impact the functional design. Also, the run-time system is directly impacted by the hardware selection(s). Therefore, late HSP could result in inadequate lead time for selecting and implementing the run-time system.

5.3 RECOMMENDATIONS FOR IMPROVING AIM

5.3.1 Narrative Descriptions of Major System Functions

Add narrative descriptions of the major system functions to the Ada Requirements Document. This would be done after completion of the AO DFD.

5.3.2 Project Management Considerations

Add the following project management considerations:

1. Reviews with customer at periodic intervals during the requirements phase (needed to ensure accuracy of requirements).

2. Reviews with customer and management at periodic intervals during requirements, design, and implementation phases (needed to ensure that project is being managed properly, within budget, on schedule, etc).

3. Constraints relating to personnel, time, money, etc. (needed in a real-world environment).

4. Implementation considerations:

   a. Development of new procedures and a user guide
   b. Development of an operations guide (run book)
   c. Development of a test plan
   d. Development of an implementation plan including a conversion plan.

5.3.3 Concurrency Modifications

These concurrency modifications are recommended for AIM:

1. Address concurrency in the non-functional requirements part of ARM instead of having a separate part for concurrency.

2. Eliminate the creation of concurrency charts during the requirements phase.

3. Use petri nets or some other tool for illustrating concurrency during design instead of the SREM-type concurrency charts.

5.3.4 Documentation of System Interfaces

Replace the N^2 Chart with a simple module interconnectivity chart that illustrates all the interfaces for each Ada design unit (i.e., subprogram, package service (entry point), or task).

Example:

Ada Design Unit	Inputs	Outputs
SUBPGM	PARM1	PARM2
PKG1.SERV1	PARM3	PARM4
•	PARM5	•
•	•	•
•	•	•
•	•	•

5.3.5 Added Emphasis on Object-Oriented Design

Make object-oriented design and information hiding the main criteria for decomposing the system into modules instead of the Structured Design methodology. However, continue to use Structured Design criteria (i.e., coupling, cohesion, and design heuristics) to evaluate the design produced via an object-oriented design methodology.

5.3.6 A-Specifications-to-Requirements Traceability

Add the development of an A-Specifications-to-Requirements traceability matrix to ARM.

5.3.7 Hardware/Software Partitioning Modifications

Hardware/software partitioning needs to begin earlier in the design process, especially during the development of distributed processing systems. Selection of hardware concomitantly with functional design will mitigate the impact of hardware selection on the functional design.

5.4 RECOMMENDATIONS FOR FURTHER RESEARCH

5.4.1 Use of Ada to Describe DFDs

Ada was used as an RSL to describe the processing requirements within each primitive function (block) of the DFDs developed during the requirements phase. However, Ada could be used to describe entire DFDs, including all the interactions between the various functions. Concurrency between the functions of a DFD could be shown via Ada's tasking construct. The Ada representation of a DFD could also be used to make corrections and automatically recreate the DFD to reflect the correction.

5.4.2 Use of Ada Earlier in the Design Process

AIM places graphic illustrations (structure charts and N^2 charts) before the Ada representation of the system during the design process. This could be reversed such that Ada specifications are used to define each major procedure or function including its name, data structures on which it operates, interfaces, calls to other functions, and a general narrative description. The Ada specifications could be used to generate a graphic illustration of the system such as a structure chart.

5.4.3 Replacement of the Current Military Project Life Cycle with an AIM-type Methodology

With the advent of Ada as the official language of the DoD, it would seem logical to change the project life cycle to make it more compatible with Ada. AIM is designed to be compatible with Ada; it makes use of packaging and tasking, and uses Ada as an RSL and a PDL. With some modifications to add more customer orientation within ARM and several project management considerations, AIM could be the basis for developing a new military project life cycle.

5.4.4 Results of Using Ada Throughout the System Development Life Cycle

The use of the implementation language to express system requirements and design is a departure from the traditional concept that requirements and design methodologies should be independent of the implementation language. From this project, it seems that the use of Ada during requirements and design is justified. However, further research is needed in this area. Specifically, Ada's use as an RSL to define the message switch requirements could be studied to determine the savings and benefits that resulted. It would also be interesting to study the reasons for and results of the decision to use the Ada RSL from requirements as the Ada PDL during design on this project.

5.4.5 Integration of Requirements and Design Methodologies for the Development of Real-Time Systems

The flow from requirements to design required a "shifting of gears" or change in mind set during the redesign of the message switch system. The requirements analysts thought of the system as a single-thread process during requirements and developed a model (DFD) that processed one entire message through the system. During design, the requirements analysts of the requirements phase became the designers, and they shifted gears almost immediately. They began to think more about the performance constraints that were identified during the requirements phase. They began designing a system structure that processed only segments of messages at a time instead of an entire message (because of performance constraints). In short, a certain kind of thinking or thought process was reserved until design. The thought process that was reserved was incompatible with the thought process that permeated the requirements phase. Therefore, the DFDs and data dictionary developed during the requirements phase were not compatible with the structure chart and data structures developed during design.

It seems that further research to determine how the problem and solution (for a real-time system) could be integrated more smoothly is appropriate. Some say that a smooth transition from requirements to design can be expected in the business world but not in a real-time environment. Still it seems that a smooth transition from requirements to design is a worthy goal regardless of the type of system being developed.

5.5 CONCLUSION

The application of AIM during the redesign of the message switch was successful. However, it is obvious from the above that many lessons were learned. Hopefully, the lessons learned and the recommendations for improvement and further research will contribute to development of a final Ada-based methodology.

BIBLIOGRAPHY

ACM. The JUG Ada Liaison Communication Newsletter, ACM - AdaTec. September, 1981.

Agerwala, Tilak. Putting Petri Nets to Work. COMPUTER. December, 1979.

Alford, M.W. Software Requirements Engineering Methodology (SREM) at the Age of Two. IEEE. 1978.

Alford, Mack W. A Requirements Engineering Method for Real-Time Processing Requirements. IEEE Transactions on Software Engineering. January, 1977, Volume SE-3, #1.

Alford, Mack W. Software Requirements Engineering Methodology (SREM) at the Age of Four. COMPSAC 80.

Andrews, Gregory. Design of Message Switching System: An Application and Evaluation of Modula. IEEE Transactions on Software Engineering. March, 1979, Volume SE-5, #2.

Baker, T.P. A One-Pass Algorithm Overloading Resolution in Ada. Math and Computer Science Dept, Florida State Univ. April, 1981.

Barnes, J.G.P. An Overview of Ada. Software--Practice and Experience. 1980, Volume 10.

Bell, Bixler, Dyer. An Extendable Approach to Computer-Aided Software Requirement Engineering. IEEE Transactions on Software Engineering. January, 1977.

Ben-Ari & Yehudai. Methodology for Modular Use of Ada. Department of Computer Science, School of Math Sciences, Tel Aviv University.

Bergland, G.D. A Guided Tour of Program Design Methodologies. Computer. October, 1981.

Berson, T.; Slegel, R.; Jacks, E.; Johnson, L.; Hershey, E.A. III; Robinson, L.; Alford, M.; Feldman, C. User Experience with Specification Tools. ACM SIGSOFT, Software Engineering Notes. July, 1979, Volume 4, #3.

Booch, Grady. Object-Oriented Design. ACM Ada Letters. March, April 1982, Volume 1, #3.

Boyd & Pizzarello. Introduction to Well Made Design Methodology. International Conference on Software Engineering, 3rd. May, 1978.

Boydstun, L.E.; Teichroew, D.; Spewak, S.; Yamancto, Y.; Starner, G. Computer Aided Modeling of Information Systems, COMPSAC 80.

Braun, Christine. Ada: Programming in the 80's. Computer. June, 1981.

Brender & Nassi. What is Ada? Computer. June, 1981.

Britton, K.; Parker, R.A.; Parnas, D. A Procedure for Designing Abstract Interfaces for Device Interface Modules. International Conference on Software Engineering, 5th.

Browne, James C. Interaction of Operating System and Software Engineering. Proceedings of the IEEE. September, 1980, Volume 68, #9.

Campos, Ivan M.; Estrin, Gerald. Concurrent Software System Design Supported by SARA at the Age of One. International Conference on Software Engineering, 3rd. May, 1978.

Campos, Ivan M.; Estrin, Gerald. SARA Aided Design Software for Concurrent Systems. AFIPS--Conference Proceedings. Volume 47.

Carlson, William E. Ada: A Promising Beginning. Computer. June, 1981.

Carlson, W.E.; Druffel, L.E.; Fisher, D.A.; Whitaker, W.A. Introducing Ada. Proceedings of the 1980 Annual Conference. October, 1980.

Chand, Donald R.; Yaday, Surya B. Logical Construction of Software. Communications of the ACM. October, 1980, Volume 23, #10.

Chung, Paul; Gaiman, Barry. Use of State Diagrams to Engineer Communications Software. International Conference on Software Engineering, 3rd. May, 1978.

Clapp, Judith A. Designing Software for Maintainability. Computer Design. September, 1981.

Clarke, L.; Wiledin, J.C.; Wolf, A.L. Nesting in Ada Programs is for the Birds. ACM SIGPLAN Notices. November, 1980, Volume 15, #11.

Comer, Douglas; Halstead, Maurice H. Simple Experiment in Top-Down Design. IEEE Transactions on Software Engineering. March, 1979, Volume SE-5, #2.

Cunha, P.R.F.; Maibaum, T.S.E. Resource = Abstract Data Type + Synchronization: A Methodology for Message Oriented Programming. International Conference on Software Engineering, 5th. 1981.

Davis, Carl G.; Vick, Charles R. The Software Development System. IEEE Transactions on Software Engineering. January, 1977.

Davis, Alan M. Automating the Requirements Phase: Benefits to Later Phases of the Software Life-Cycle. COMPSAC 80.

Davis, A.M.; Miller, T.J.; Rhode, E.; Taylor, B.J. RLP: An Automated Tool for the Processing of Requirements. COMPSAC 79.

Dewolf, J. Barton. Requirements Specification and Preliminary Design for Real-Time Systems. The Charles Stark Draper Laboratory, Inc. 1977.

Distaso, Jack R. A Software Management-Survey of the Practice in 1980. Proceedings of the IEEE. September, 1980, Volume 68, #9.

Estrin, Gerald. A Methodology for Design of Digital Systems--Supported by SARA at the Age of One. AFIPS--Conference Proceedings. Volume 47.

General Dynamics/Data Systems Division. Requirements Documentation Automatic Direct Flow Graphs. SES-RD-008. September, 1980.

Geschke, Charles M.; Morris, James H., Jr.; Satterwaite, Edwin H. Early Experience with Mesa. Communications of the ACM. August, 1977, Volume 20, #8.

Goodenough, John. The Ada Compiler Validation Capability. Computer. June, 1981.

Goodenough, John. Ada Compiler Valid Implementers Guide. SofTech. October, 1980.

Griffiths, S.N. Design Methodologies - A Comparison. Infotech State of the Art Report.

Guttag, J.V.; Horning, J.J. The Algebraic Specification of Abstract Data Types. ACTA Information. 1978, Volume 10, pp. 27-52.

Guttag, John. Abstract Data Types and the Developmental Data Structures. Communications of the ACM. June, 1977, Volume 20, #6.

Haberman, A.N. The Use of Ada Packages. Using Selected Features of Ada: A Collection of Papers. Center for Tactical Computer Systems, U.S. Army Communication-Electronics Command, Fort Monmouth, N.J. 1980.

Hamilton, M.; Zeldin, S. The Relationship Between Design and Verification. The Journal of Systems and Software. 1979, 1.29-56.

Hamilton, M.; Zeldin, S. Higher Order System Software (HOS) Concepts. HOS, Inc. June, 1976.

Hamilton, M. A Discussion of Higher Order Software Concepts as they apply to Functional Requirements and Specifications. Charles Stark Draper Laboratory. December, 1973.

Hart, Hal. Ada for Design: An Approach for Transitioning Industry Software Developers. For Presentation at NSIA Software Group Conference. October, 1981.

Hebalkar, P.G.; Zilles, S.N. TELL: A System for Graphically Representing Software Designs. IBM Research Laboratory--San Jose. October, 1978.

Heninger, Kathryn L. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. IEEE Transactions on Software Engineering. January, 1980, Volume SE-6, #1.

Higgins, David A. Program Design and Construction. Prentice-Hall, Inc., Englewood Cliffs, N.J. 1979.

Honig, William L.; Carlson, C. Robert. Toward Understanding of (Actual) Data Structures. The Computer Journal. 1976, Volume 21, #2.

Heubner, Doug L. System Validation through Automated Requirements Verification. COMPSAC 79.

ISDOS. PSA Modifier Commands. ISDOS - Department of Industrial and Operations Engineering, University of Michigan.

ISDOS. PSA Command Summary. ISDOS - Department of Industrial and Operations Engineering, University of Michigan.

ISDOS. PSA Utility & Stand-Alone Programs. ISDOS - Department of Industrial and Operations Engineering, University of Michigan.

ISDOS. PSA Error Messages and Diagnostics. ISDOS - Department of Industrial and Operations Engineering, University of Michigan.

ISDOS. PSL Language Reference Summary ISDOS - Department of Industrial and Operations Engineering, University of Michigan.

ISDOS. PSL Users Manual. ISDOS - Department of Industrial and Operations Engineering, University of Michigan. March, 1975.

ISDOS. PSA Reports and Report Commands. ISDOS - Department of Industrial and Operations Engineering, University of Michigan.

Jackson, M.A. Information Systems: Modelling, Sequencing, and Transformations. International Conference on Software Engineering, 3rd. May, 1978.

Jackson, M.A. Constructive Methods of Program Design. Conference of the European Cooperation in Informatics. August, 1976.

Kenyon, Ralph E. Little Ada (Part I) Microsystems 2.5. September/October 1981, Volume 2, #5.

Klos, L.C.; Snodgrass, J.G. (Collection of Papers on Directed Flowgraphs). General Dynamics/Data Systems Division Internal Drafts. 1980.

Klos, Larry C. An Interface Management Approach to Software Development. General Dynamics/Data Systems Division.

Komoda, N.; Haruna, K.; Kaji, H.; Shinozawa, H. An Innovative Approach to System Requirements Analysis by Using Structural Modelling Methods. International Conference on Software Engineering, 5th. March, 1981.

Lano, R. J. The N2 Chart. TRW. November, 1977.

Larson, R.E.; Berman, L.E.; Steding, T.L.; Hilborn, G. Development of a Unified Approach for System Requirements Engineering. COMPSAC 79.

Ledgard, Henry F.; Taylor, Robert W. Two Views of Data Abstraction. Communications of the ACM. June, 1977, Volume 20, #6.

Ledgard, Henry. Ada: An Introduction and Ada Reference Manual. Springer-Verlag, N.Y., N.Y. 1981.

Levitt, K.N.; Neuman, P.; Robinson, L. The SRI Hierarchical Development Methodology (HDM) and Its Application to the Development of Secure Software. U.S. Dept. of Commerce, National Bureau of Standards, Special Publication 500-67. October, 1980.

Linden, Theodore. The Use of Abstract Data Types to Simplify Program Modification. National Bureau of Standards.

Liskov, B.; Snyder, A.; Atkinson, R.; Schaffert, C. Abstraction Mechanisms in CLU. Communications of the ACM. August, 1977, Volume 20, #8.

Loveman, David B. Tutorial On Ada Exceptions. Using Selected Features of Ada: A Collection of Papers. Center for Tactical Computer Systems, U.S. Army Communication-Electronics Command, Fort Monmouth, N.J.

Miller, Raymond E.; Kasai, Takumi. Comparing Models of Parallel Computation by Homomorphisms. COMPSAC 79.

Miller, Raymond E. A Comparison of Some Theoretical Models of Parallel Computation. IEEE Transactions on Software Engineering. August, 1973, Volume 22, #8.

Morris, Alfred H. Can Ada Replace Fortran for Numerical Computation? Naval Surface Weapons Center. 1981.

Myers, Glenford J. Software Reliability Principles and Practices. John Wiley & Sons, N.Y. 1976.

Myers, Glenford J. Composite/Structured Design. Van Nostrand Reinhold Company. N.Y., N.Y. 1978.

Nestor, John. Types. Using Selected Features of Ada: A Collection of Papers. Center for Tactical Computer Systems, U.S. Army Communication-Electronics Command, Fort Monmouth, N.J.

Ohba, M.; Kodota, H.; Tanetsu, Y.; Takimoto, N. Architecture Kernel: Higher Level Program Specification. COMPSAC 80.

Orr, Kenneth T. Structured Systems Development. Yourdon Press. N.Y., N.Y. 1977.

Orr, Kenneth T. Introducing Structured Program Design. Infotech State of the Art Report.

Page-Jones, Meilir. The Practical Guide to Structured Systems Design. Yourdon Press. N.Y., N.Y. 1980.

Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM. December, 1972.

Parnas, D.L. On the Design and Development of Program Families. IEEE Transactions on Software Engineering. March, 1976.

Parnas, D.L. Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering. March, 1979, Volume SE-5, #2.

Patterson, John C. Real-Time Multiprocessing May Depend on Ada Software. Defense Electronics. August, 1981.

Penedo, Maria Heloisa; Berry, Daniel M. The Use of a Module Interconnection Language in the SARA System Design Methodology. International Conference on Software Engineering, 4th. 1979.

Perry, DeWayne. Low Level Language Features. Using Selected Features of Ada: A Collection of Papers. Center for Tactical Computer Systems, U.S. Army Communication-Electronics Command, Fort Monmouth, N.J.

Peters, Lawrence L.; Tripp, Leonard L. Comparing Software Design Methodologies. Datamation.

Peters, Lawrence. Software Representation and Composition Techniques. Proceedings of the IEEE. September, 1980, Volume 68, #9.

Peterson, J.E. Data State Design. IBM COMPCON. 1976.

Peterson, James L. Petri Nets. Computing Surveys. September 1977, Volume 9, #3.

Peterson, James L. Petri Net Theory and the Modelling of Systems. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1981.

Razouk, R.R.; Vernon, M.; Estrin, G. Evaluation Methods in SARA--The Graph Model Simulator. Computer Science Department, UCLA.

Razouk, R.R.; Estrin, G. Validation of the X.21 Interface Specification Using SARA. Computer Science Department, UCLA.

Reifer, Donald J. Comparative Software Engineering. Reifer Consultants, Inc.

Reinstein, H.C.; Hollander, C.R. A Knowledge-Based Approach to Applications Development for Non-Programmers. IBM, Palo Alto. July, 1979.

Riddle, William E.; Wileden, Jack C. Languages for Representing Software Specifications and Design. ACM SIGSOFT. October, 1978, pp. 7-11.

Riddle, William E. An Event Based Design Methodology Supported by Dream. Formal Models and Practical Tools for Information Systems Design. 1979.

Roman, Gruia-Catalin. Verification Procedures Supporting Software Systems Development. National Computer Conference. 1979.

Rosenbaum, Jacob D.; Kutin, Edward B. A Microprocessor-Based Front-End Analysis & Modeling Environment. High Order Software, Inc. April, 1980.

Ross, Douglas T.; Shoman, Kenneth E., Jr. Structured Analysis for Requirements Definition. IEEE Transactions on Software Engineering. January, 1977, Volume SE-3, #1.

Ross, Douglas T. Structured Analysis (SA): A Language for Communicating Ideas. IEEE Transactions on Software Engineering. January, 1977, Volume SE-3, #1.

Ruggiero, W.; Estrin, G.; Fenchel, R.; Razouk, R.; Schwabe, D.; Vernon, M. Analysis of Data Flow Models Using the SARA Graph Model of Behavior. AFIPS--Conference Proceedings. Volume 48.

Schindler, Max. Compcon Speakers Analyze Productivity in Software. Electronic Design. September, 1981.

Schultz, Brad. Push Single, Standardized Ada Backed. Computerworld. November, 1981.

Schwartz, J.T. Automatic Data Structure Choice in a Language of Very High Level. Communications of the ACM. December, 1975, Volume 18, #12.

Scott, Leighton R. An Engineering Methodology for Presenting Software Functional Architecture. International Conference on Software Engineering, 3rd. May, 1978.

Shaw, M.; Wulf, W.A. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. Communications of the ACM. August, 1977, Volume 20, #8.

Shaw, Mary. The Impact of Abstraction Concerns on Modern Programming Languages. Proceedings of the IEEE. September, 1980, Volume 68, #9.

Shuman, Steven. Tutorial on Ada Tasking. Using Selected Features of Ada: A Collection of Papers. Center for Tactical Computer Systems, U.S. Army Communications-Electronics Command, Fort Monmouth, N.J. March, 1981.

Snodgrass, Jerry. A View on the Usefulness of Algebraic Abstract Data Types During the Software System Life Cycle. General Dynamics/Data Systems Division, Central Center. March, 1979.

Snodgrass, Jerry. On Providing Implementations of Abstract Data Types to Programmers. General Dynamics/Data Systems Division, Central Center. April, 1979.

SofTech. Draft ALS ROLM 1666 Code Generator B5. HQ US Army CECOM, Fort Monmouth, N.J. August, 1981.

SofTech. (Data Diagrams). SofTech 9022-78.

SofTech. Draft ALS Command Language Processor B5 Specification. HQ US ARMY CECOM, Fort Monmouth, N.J. August, 1981.

SofTech. Draft ALS ROLM 1602B Code Generator B5 Specification. HQ US ARMY CECOM, Fort Monmouth, N.J. August, 1981.

SofTech. ALS VAX-11/780 Assembler B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. July, 1981.

SofTech. ALS VAX-11/780 VAX/VMS Runtime Support Library. HQ U.S. Army CECOM, Fort Monmouth, N.J. July, 1981.

SofTech. ALS PDP-11/70 Unix Linker B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. June, 1981.

SofTech. ALS RCLM 1666 Assembler B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. July, 1981.

SofTech. ALS ROLM 1602B Assembler B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. July, 1981.

SofTech. ALS Compiler Machine-Independent Section B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. May, 1981.

SofTech. ALS Bare VAX-11/780 Runtime Support Library. HQ U.A. Army CECOM, Fort Monmouth, N.J. July, 1981.

SofTech. ALS ROLM Linker B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. June, 1981.

SofTech. ALS Bare VAX-11/780 Loader B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. May, 1981.

SofTech. ALS VAX-11/780 Code Generator B5 Specification HQ U.S. Army CECOM, Fort Monmouth, N.J. June, 1981.

SofTech. ALS VAX-11/780 Linker B5 Specification. HQ U.S. Army CECOM, Fort Monmouth, N.J. June, 1981.

SofTech. ALS Overview. HQ U.S. Army CECOM, Fort Monmouth, N.J.

Spitzen, Jay M.; Levitt, Karl N.; Robinson, Lawrence. An Example of Hierarchical Design and Prcof. Communications of the ACM. December, 1978, Volume 21, #12.

Spitzen, Jay; Wegbreit, Ben. The Verification and Synthesis of Data Structures. ACTA Informatica. 1975, Volume 4, #127.

Stenning, Vic; Froggatt, Terry; Gilbert, Roger; Thomas, Ellis. The Ada Environment: A Perspective. Computer. June, 1981, p.17.

Stephens, Sharon A.; Tripp, Leonard L. Requirements Expression and Verification Aid. International Conference on Software Engineering, 3rd. May, 1978.

Stevens, W.P.; Myers, G.J.; Constantine, L.L. Structured Design. IBM Systems Journal. 1974, Volume 13, #2.

Sutton, Steven A.; Basili, Victor R. The Flex Software Design System: Designers Need Languages, Too. Computer. November, 1981.

Tausworthe, Robert. Standardized Development of Computer Software. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1977.

Teichroew, Daniel; Hershey, Ernest A, III. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. IEEE Transactions on Software Engineering. January, 1977, Volume SE-3, #1.

Teichroew, Daniel. A Survey of Languages for Stating Requirements for Computer-Based Information Systems. Fall Joint Computer Conference. 1972.

Telplitzky, Philip. An Approach for Choosing a Programming Specification Methodology. COMPSAC 79.

Trattnig, W; Kerner, H. EDDA, A Very-High-Level Programming and Specification Language in the Style of SADT. COMPSAC 80.

TRW. Requirements Networks. Ballistic Missile Defense Advanced Technology Center. 1976.

TRW. Software Requirements Engineering Methodology (SREM) Applied as a Software Management System. Excerpts from a SREM Presentation.

Unknown. Ada Program Designs. General Dynamics-Re:F-12101. December, 1980.

Unknown. Making Software Right the First Time. ICP Interface Administrative & Accounting. Spring 1977.

Unknown. Two Pairs of Examples in the Jackson Approach to System Development. Unknown.

Van Deusen, Mary. Types in Red. Prime Computer Inc.

Voss, Klaus. Using Predicate/Transition-Nets to Module and Analyze Distributed Database Systems. COMPSAC 79.

Warnier, Jean-Dominique. Logical Construction of Systems. Van Nostrand Reinhold Company, N.Y., N.Y. 1981.

Wasserman, A. I. Information System Design Methodology. Journal of the American Society for Information Sciences. January, 1980, Volume 32, #1.

Waugh, D. W. Ada as a Design Language. IBM Software Engineering Exchange. October, 1980, Volume 3, #1.

Wheeler, Thomas J. Embedded System Design with Ada System Design Language. Report/Ada Prebidders Conference. November, 1980.

Wichmann, Brian. Real Data Types in Ada. Using Selected Features of Ada: A Collection of Papers. Center for Tactical Computer Systems, U.S. Army Communication-Electronics Command, Fort Monmouth, N.J.

Winters, Edward W. An Analysis of the Capabilities of Problem Standard Language: A Language for System Requirements and Specifications. COMPSAC 79.

Wolfe, Martin I.; Babich, Wayne; Simpson, Richard; Thall, Richard; Weissman, Larry. The Ada Language System. Computer. June, 1981.

Yeh, Raymond T. Current Trends in Programming Methodology. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1977.

Yeh, Raymond T.; Zave, Pamela; Conn, Alex Paul; Cole, George E., Jr. Software Requirements: A Report on the State of the Art. Technical Report-949. University of Maryland. Cctober, 1980.

Yourdon, Edward; Constantine, Larry L. Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Yourdon Press. N.Y., N.Y. 1978.

Yourdon, Edward. Techniques of Program Structure and Design. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1975.

Zave, Pamela; Yeh, Raymond T. Specifying Software Requirements. Proceedings of the IEEE. September, 1980, Volume 69, #9.

Zave, Pamela; Yeh, Raymond T. Executable Requirements for Embedded Systems. International Conference on Software Engineering, 5th. March, 1981.

Zave, Pamela. The Operational Approach to Requirements Specifications for Embedded Systems. Technical Report-976. University of Maryland. December, 1980.

Zave, Pamela. A Comprehensive Approach to Requirements Problems. COMPSAC 79.

Zave, Pamela. Functional Specification of Asynchronous Processes and Its Application to the Early Phases of System Development. Technical Report-775. .University of Maryland. March, 1979.

Zeigler, Stephen; Allegre, Nicole; Johnson, Robert; Morris, James; Burner, Gregory. Ada for the Intel 432 Microcomputer. Computer. June, 1981.